HIGH-PERFORMANCE ALGORITHMS AND SOFTWARE FOR LARGE-SCALE MOLECULAR SIMULATION

A Thesis
Presented to
The Academic Faculty

by

Xing Liu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology
May 2015

HIGH-PERFORMANCE ALGORITHMS AND SOFTWARE FOR LARGE-SCALE MOLECULAR SIMULATION

Approved by:

Professor Edmond Chow,
Committee Chair
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Edmond Chow, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Professor David A. Bader
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Professor C. David Sherrill
School of Chemistry and Biochemistry
Georgia Institute of Technology

Professor Jeffrey Skolnick
Center for the Study of Systems Biology
Georgia Institute of Technology

Date Approved: 10 December 2014

To my wife, Ying Huang

the woman of my life.

ACKNOWLEDGEMENTS

I would like to first extend my deepest gratitude to my advisor, Dr. Edmond Chow, for his expertise, valuable time and unwavering support throughout my PhD study. I would also like to sincerely thank Dr. David A. Bader for recruiting me into Georgia Tech and inviting me to join in this interesting research area.

My appreciation is extended to my committee members, Dr. Richard Vuduc, Dr. C. David Sherrill and Dr. Jeffrey Skolnick, for their advice and helpful discussions during my research. Similarly, I want to thank all of the faculty and staff in the School of Computational Science and Engineering at Georgia Tech. I am also thankful for the support and guidance of Dr. Mikhail Smelyanskiy and Dr. Pradeep Dubey when I was interning at Intel Labs.

To my friends at Georgia Tech, Jee Choi, Kent Czechowski, Marat Dukhan, Oded Green, Adam McLaughlin, Robert McColl, Lluis Miquel Munguia, Aftab Patel, Piyush Sao, Zhichen Xia, and Zhaoming Yin, thanks for helping me in many ways and their inspiring discussions on research (and other things).

Finally, and most importantly, my recognition goes out to my family, specially to my beloved wife, Ying Huang, for their support, encouragement and patience during my pursuit of PhD. Thanks for their faith in me and allowing me to be as ambitious as I wanted. Without their constant support and encouragement, I would not have completed the PhD program.

Contents

List of Tables

List of Figures

SUMMARY

Molecular simulation is an indispensable tool in many different disciplines such as physics, biology, chemical engineering, materials science, drug design, and others. Performing large-scale molecular simulation is of great interest to biologists and chemists, because many important biological and pharmaceutical phenomena can only be observed in very large molecule systems and after sufficiently long time dynamics. On the other hand, molecular simulation methods usually have very steep computational costs, which limits current molecular simulation studies to relatively small systems. The gap between the scale of molecular simulation that existing techniques can handle and the scale of interest has become a major barrier for applying molecular simulation to study real-world problems.

In order to study large-scale molecular systems using molecular simulation, it requires developing highly parallel simulation algorithms and constantly adapting the algorithms to rapidly changing high performance computing architectures. However, many existing algorithms and codes for molecular simulation are from more than a decade ago, which were designed for sequential computers or early parallel architectures. They may not scale efficiently and do not fully exploit features of today's hardware. Given the rapid evolution in computer architectures, the time has come to revisit these molecular simulation algorithms and codes.

In this thesis, we demonstrate our approach to addressing the computational challenges of large-scale molecular simulation by presenting both the high-performance algorithms and software for two important molecular simulation applications: Hartree-Fock (HF) calculations and hydrodynamics simulations, on highly parallel computer architectures. The algorithms and software presented in this thesis have been used by biologists and chemists

to study some problems that were unable to solve using existing codes. The parallel techniques and methods developed in this work can be also applied to other molecular simulation applications.

Chapter I

INTRODUCTION

1.1 Background

Molecular simulation encompasses a set of computational methods for simulating the behavior and properties of molecules. These methods can be used by biologists and chemists to reveal microscopic details that are difficult to observe by experimental or theoretical approaches [76], Molecular simulation has a myriad of applications in multiple disciplines including biology, chemical engineering, drug design and materials science.

Molecular simulation as a technique dates back to 1950s, immediately after electronic computers became available for nonmilitary use [46]. The first study of molecular simulation was carried out in 1953 by Metropolis et al. [80], who used the Monte Carlo method to study the properties of a liquid. Another pioneering work was performed in 1955 by Fermi et al. [40], who simulated the dynamics of an one-dimensional crystal. At that time, very few research scientists have access to computers and know how to use computers to simulate a molecular system. In fact, the first studies of molecular simulation were partially motivated by the desire to evaluate the logical structure and demonstrate the capabilities of the MANIAC computer at Los Alamos [46].

Today, using computers to simulate a molecular systems has become standard practice, and molecular simulation mainly serves a twofold purpose. First, it can be used to predict the properties of molecules. Before molecular simulation was invented, the only way to study a molecular system was to make use of a molecular theory to provide an approximate description of that molecular system [46]. This approach usually cannot give accurate results because very few molecular theories can be expressed in terms of equations that we can solve exactly. Using molecular simulation methods, however, can provide the results to

any desired accuracy. Another important application of molecular simulation is to validate theories. For any given molecular theory, one can build a modeling system, and molecular simulation can be used to obtain the exact results of the modeling system. We can now compare the results of molecular simulation to experiment. We know that the theory is flawed, if we find that experimental results and simulation results disagree. Nowadays it is becoming increasingly common that a molecular theory is tested by molecular simulation before it is applied to the real world.

The desire for a theoretical understanding of molecular systems has motivated the invention of a considerable number of molecular simulation methods, such as Monte Carlo methods, free energy methods, and others. In general, those methods can be divided into separate areas, distinguished by the length scale and the models that are used [19]. Here, we will briefly describe three most important areas. First, on the atom scale, we can explicitly model electrons of each atom. This is the area of *quantum chemistry* (QC). By explicitly modeling the electronic structure of a molecule, almost of all of the molecule's chemical and physical properties can be predicted. QC is considered the most accurate molecular simulation method, although its calculations are usually very computationally expensive,

At the next level of the molecular scale, one can treat atoms as the smallest individual unit and model them as spherical particles. Instead of using quantum physics, Newton's equations of motion are solved to calculate the interactions and motion of particles. The simulation methods that use this modeling scheme fall into the area of *molecular dynamics* (MD). MD was invented in 1956 by Alder and Wainwright [3], who used MD to study the dynamics of an assembly of hard spheres. Almost at the same time, Rahman [96] used MD to study a real liquid (Argon). Compared to QC, MD methods are less computationally expensive and can be applied to larger molecular systems.

At last, on the even larger length scale of macromolecules, simulating any reasonable sized molecular system using QC or MD becomes too expensive to compute. To study such a molecular system, coarse-grain (CG) models are required. One particular CG

model was designed for simulating macromolecules in a fluid environment, which is called *Brownian/Stokesian dynamics* (BD/SD). In BD/SD simulations, only solute molecules are treated explicitly; solvent molecules are modeled implicitly as random forces on the solute molecules. The BD and SD methods appeared on the scene in 1970s. The first studies were reported by Ermak and McCammon [38], and by Dickinson [29].

Although there are many different types of molecular simulation methods, these methods encounter a large number of common computational problems. Table 1 lists these common computational problems. In this thesis, we will demonstrate our solutions to these computational problems by presenting the high-performance algorithms and software for two important molecular simulation applications: Hartree-Fock (HF) calculations and Brownian/Stokesian dynamics (BD/SD) simulations. We chose these two applications not only because they are very important scientific applications but also because most of the common computational problems in molecular simulation can be found in them. In the following of this section, we will give a brief overview on HF and BD/SD.

Table 1: Common computational problems in molecular simulation.

Computational problems	Simulation methods
Sparse matrix multiplication	BD, SD
FFTs	BD, SD, MD
N-body problem	BD, SD, MD, QC
Long-range interactions	BD, MD
Short-range interactions	SD, MD
Iterative solve	SD, QC
Computing Brownian displacements	BD, SD

The Hartree-Fock method, also known as the self-consistent field (SCF) method, is central to quantum chemistry [109]. It approximately solves the electronic Schrödinger equation and is a valuable method for providing a qualitative description of the electronic structure of molecular systems. More importantly, the HF method is also the starting point for more sophisticated and accurate electronic structure methods that include coupled cluster theory and many-body perturbation theory [12].

Brownian dynamics (BD) is a computational method for simulating the motion of particles, such as macromolecules and nanoparticles, in a fluid environment. It has a myriad of applications in multiple areas including biology, biochemistry, chemical engineering and materials science [79, 31]. In biology, BD is used to study the motions and interactions of proteins, DNA, and other biologically important molecules. In chemical engineering, BD is used to study the properties of industrially important colloidal materials and complex fluids. For macromolecules in solvent, it is important to accurately model the hydrodynamic interactions (HI), that is, the forces mediated by the solvent on one particle due to the motion of other particles. Hydrodynamic forces are long range, varying as $1/\|\vec{r}\|$, where $\vec{r}$ is the inter-particle separation. The modeling of HI is essential for correctly capturing the dynamics of particles, particularly collective motions [60, 66].

Although BD has been a staple technique, it is not able to accurately model HI in high volume fraction systems, such as the crowded environment inside biological cells. This is because the RPY tensor used in the BD algorithm makes the assumption that the particles are widely separated. To model the situation where particles may be nearly touching, there is no appropriate tensor that is positive definite. A solution is to use Stokesian dynamics (SD) [18, 19, 36], which was first developed in chemical engineering. SD is able to accurately models both long- and short-range hydrodynamic forces, in contrast to the BD method which cannot accurately model short-range forces, This capability, however, makes SD much more computationally demanding than BD.

Like in BD simulations, the macromolecules in SD are modeled as spherical particles of possibly varying radii. The macromolecules may be colloids, polymers, proteins, or other macromolecules in environments where the inertial forces are much smaller than the inter-particle forces, i.e., the particle Reynolds number is small. At each time step, like in other particle simulation methods, forces on the particles are computed and then the particle positions are updated.

The significance of SD for biologists is its capability to model the crowded environment

inside the cell, a condition that has only recently become appreciated [37, 74, 124, 125]. SD has been recently used to simulate the *E. coli* cytoplasm [7], and attracting significant interest [32].

1.2 Motivation

Our motivation is to use high performance computing systems and techniques to enable large-scale molecular simulation applications. In molecular simulation contexts, the term "scale" are used for both length and time. Performing a large length scale and time scale molecular simulation is of great interest to biologists and chemists, because many important biological and pharmaceutical phenomena can only be observed in very large molecule systems and after sufficiently long time dynamics.

While molecular simulation is considered an indispensable tool in a growing number of disciplines, its methods usually have very steep computational costs, which limits current molecular simulation studies to relatively small systems. Thus, there exists a gap between the scale of molecular simulation that existing techniques can handle and the scale of interest. This has become a major barrier for adopting new molecular simulation applications. In order to study large-scale molecular systems using molecular simulation, it requires developing highly parallel simulation algorithms and constantly adapting the algorithms to rapidly changing high performance computing architectures.

On the other hand, many existing algorithms and codes for molecular simulation are from more than a decade ago, which were designed for sequential computers or early parallel architectures [53, 44, 33, 47]. They may not scale efficiently and do not fully exploit features of today's hardware. Given the rapid evolution in computer architectures, the time has come to revisit these molecular simulation algorithms and codes.

This thesis is partially motivated by the increasing use of heterogeneous computing systems, which are designed for highly parallel workloads, but are challenging to utilize

efficiently. The common challenges include: 1) optimization of sparse and irregular algorithms, 2) adapting memory-hungry applications for hardware that has relatively low memory capacities, 3) exploiting vectorization for calculations that do not naturally possess SIMD parallelism, and 4) load-balancing between CPUs and accelerators. As such, it is not clear to what extent a molecular simulation applications might benefit from heterogeneous computing and how to effectively design and engineering an efficient implementation.

Chapter II

SCALABLE DISTRIBUTED PARALLEL ALGORITHMS FOR FOCK MATRIX CONSTRUCTION

Fock matrix construction is the computationally dominant step of Hartree-Fock calculations. It also arises in Density Functional Theory (DFT) [94], another highly accurate and widely used method for electronic structure computations. An efficient Fock matrix construction program is an essential part of a quantum chemistry program suite, and in this chapter we will consider the problem of parallel Fock matrix construction on distributed systems. We will first provide essential background for Fock matrix construction in the context of Hartree-Fock calculations, followed by a discussion of the involved main computational challenges and the limitations of the previous work. We will then present a new scalable distributed parallel algorithm for Fock matrix construction that successfully addresses its computational challenges. We will also describe a heterogenous implementation of the new algorithm using the Intel Xeon Phi coprocessor.

2.1 Background: Hartree-Fock Method and Fock Matrix Construction

2.1.1 Hartree-Fock Equations

Given a set of atomic positions for a molecule, the Hartree-Fock (HF) method approximately solves the electronic Schrödinger equation for a molecule with a given number of electrons,

$$\hat{H}_{elec}\Psi_{elec} = E\Psi_{elec}$$

In this expression Ψ_{elec} is called a *wavefunction*, E is the energy of the wavefunction, and $\hat{H}_{elec}$ is a given differential operator known as the *electronic Hamiltonian*. The quantum

mechanical information about the molecule is contained in its wavefunction, and almost all the molecule's chemical and physical properties can be predicted using Ψ_{elec}.

For a system with n_{el} electrons the wavefunction Ψ_{elec} can be expressed as [61]

$$\Psi_{elec} = \begin{pmatrix} \psi_1(x_1) & \psi_2(x_1) & \dots & \psi_{n_{el}}(x_1) \\ \psi_1(x_2) & \psi_2(x_2) & \dots & \psi_{n_{el}}(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_1(x_{n_{el}}) & \psi_2(x_{n_{el}}) & \dots & \psi_{n_{el}}(x_{n_{el}}) \end{pmatrix}$$

where ψ_i represents a *molecular orbital*, and x_k denotes the spatial and spin coordinates of electron k. In the HF method, molecular orbitals are expressed in terms of a set of *basis functions* ϕ_j,

$$\psi_i(\mathbf{r}) = \sum_{j=1}^{n_f} c_{ij}\phi_j(\mathbf{r}) \tag{1}$$

where n_f is the number of the basis functions, and c_{ij} is a *molecular orbital coefficient*. The HF numerical procedure determines the coefficients c_{ij} and the energy associated with the computed wavefunction. In practice, to begin the computation, a *basis set* is chosen, which is a set of basis functions for each atom in the molecule. Basis sets with more basis functions per atom lead to more accurate approximate solutions than those with fewer basis functions. The computational cost of the HF calculation is directly related to the number of basis functions used to represent the solution.

Expressing the electronic Schrödinger equation using Equation (1) yields a nonlinear generalized eigenvalue problem, the Roothaan equations [61],

$$FC = SC\varepsilon$$

where C is a matrix containing the coefficients c_{ij}, F is called the *Fock matrix* and depends on C (this dependence makes the eigenvalue problem nonlinear), S is a fixed matrix called the *overlap matrix* that depends on the basis set, and ε is a diagonal matrix of eigenvalues or energies. All matrices are of size $n_f \times n_f$. The generalized nonlinear eigenvalue problem

is transformed to a standard nonlinear eigenvalue problem by using a basis transformation, $X = Us^{1/2}$, where UsU^T is the eigenvalue decomposition of S.

In the above, the Fock matrix is constructed using the formula

$$\begin{aligned} F_{ij} &= H_{ij}^{core} + G_{ij}(D) \\ &= H_{ij}^{core} + \sum_{kl} D_{kl} \left(2(ij|kl) - (ik|jl)\right) \end{aligned}$$

Here, H^{core} is called the *core Hamiltonian matrix*, $(ij|kl)$ represents an *electron repulsion integral* (ERI), and D is the *density matrix*, which can be computed from the molecular orbital coefficients,

$$D = C_{occ} C_{occ}^T$$

where C_{occ} is the matrix formed by the columns of C corresponding to the smallest $n_{el}/2$ eigenvalues of $X^T F X$, where n_{el} is the number of electrons of the molecule.

Given F and D, the electronic energy of the molecule is computed as

$$E_{el} = \frac{1}{2} \sum_{ij} D_{ij} (H_{ij}^{core} + F_{ij})$$

and the total HF energy is the sum of the nuclear repulsion energy and the electronic energy.

2.1.2 Hartree-Fock Algorithm

The Hartree-Fock algorithm is an iterative method for computing the molecular orbital coefficients c_{ij}. At each iteration, the Roothaan equations are solved for C and ε. It takes, as its input, the coordinates, in $\mathbb{R}^3$, of the atoms of the molecule, the atomic numbers of the atoms, a basis set, and an initial guess. As its output, it produces three matrices, a Fock matrix F, a density matrix D, and a coefficient matrix C.

Each iteration k of the HF algorithm has two main computational steps, the formation of the k-th approximation to the Fock matrix F, using the $(k-1)$-th density matrix D, and the diagonalization of this matrix to obtain the k-th approximation to D. The iteration is usually stopped when the magnitude of the difference in values between the current density

matrix and the previous density matrix has fallen below a certain, pre-chosen, convergence threshold. This is the origin of the term "self-consistent field."

Algorithm 1: Hartree-Fock Algorithm

```
Guess D
Compute H^core
Diagonalize S = UsU^T
Form X = Us^{1/2}
repeat
    Construct F = H^core + G(D)
    Form F' = X^T F X
    Diagonalize F' = C' ε C'^T
    C = X C'
    Form D = 2 C_occ C_occ^T
until converged
```

The HF algorithm is presented in Algorithm 1. The matrices S, X, and H_{core} do not change from one iteration to the next and are usually precomputed and stored.

2.1.3 Electron Repulsion Integrals

The construction of the Fock matrix involves a four dimensional array of numbers, each of whose entries is denoted by $(ij|kl)$ where the indices i, j, k, l run from 0 to $n_f - 1$, where n_f is the number of basis functions. Each $(ij|kl)$ is a six dimensional integral, called an electron repulsion integral (ERI), given by

$$(ij|kl) = \int \phi_i(x_1)\phi_j(x_1)r_{12}^{-1}\phi_k(x_2)\phi_l(x_2)dx_1dx_2 \tag{2}$$

where the x_1 and x_2 are coordinates in $\mathbb{R}^3$, $r_{12} = \|x_1 - x_2\|$, and the integral is over all of $\mathbb{R}^3 \times \mathbb{R}^3$.

The ERIs possess permutational symmetries, given by

$$(ij|kl) = (ji|kl) = (ij|lk) = (ji|lk) = (kl|ij) \tag{3}$$

Hence, the number of unique ERIs is $\approx n_f^4/8$. It is prohibitively expensive to precompute and store the ERIs in memory for all but the smallest of molecules; they must be

recomputed each time a Fock matrix is constructed, which is once per iteration of the HF algorithm. ERI computation is expensive, making Fock matrix construction a significant portion of the runtime of the HF algorithm.

To understand the computation of ERIs, which depend on the basis functions, we must know that basis functions are divided into groups called *shells*, which vary in size. To define these, first note that a primitive basis function centered at the origin has the form

$$\phi(r) = x^{m_x} y^{m_y} z^{m_z} \exp(-\alpha r^2)$$

where $r = (x, y, z) \in \mathbb{R}^3$. The parameters of the basis functions are the nonnegative integers m_x, m_y, m_z, and the Gaussian exponent α. Shells are groups of basis functions with the same center and same *angular momentum* $L_a = m_x + m_y + m_z$. For angular momentum 0, 1, 2, 3, the shells are called s, p, d, f, and contain 1, 3, 6, 10 basis functions, respectively.

An important optimization in the computation of ERIs is to compute them in batches called *shell quartets*, defined as

$$(MN|PQ) = \{(ij|kl) \text{ s.t. } i \in \text{shell } M,\ j \in \text{shell } N,\ k \in \text{shell } P,\ l \in \text{shell } Q\}.$$

These batches are 4-dimensional arrays of different sizes and shapes. The fact that these irregular batches are the minimal units of work is the main reason for the great complexity of efficient parallel HF codes.

2.1.4 Screening

It turns out that many of the $n_f^4/8$ ERIs are zero or negligibly small. This is a consequence of the fact that the integral in Equation (2) is small if the centers of the pair of the ϕ to the left or right of r_{12} are centered at points in space that are physically far from each other. More exactly, we have the relation [107]

$$(ij|kl) \leq \sqrt{(ij|ij)(kl|kl)}$$

Thus, if we have determined a drop tolerance τ for $(ij|kl)$ that yields the required accuracy of the computed F and D from the HF calculations, we can neglect the computation of the integrals $(ij|kl)$ for which

$$\sqrt{(ij|ij)(kl|kl)} < \tau. \tag{4}$$

The use of relation (4) to drop integrals is a procedure called *Cauchy-Schwarz screening*, and it can be shown that the number of integrals remaining after applying screening is significantly less than $n_f^4/8$, especially for large molecules [107]. Thus, for computational efficiency, it is essential to utilize screening.

Since integrals are computed in batches called shell quartets, we require a few definitions to illustrate how screening is applied to shell quartets. The *shell pair value* of a pair of shells is

$$s(MN) = \max_{i \in M\, j \in N} (ij|ij)$$

In practice, the shell pair values are usually computed and stored. Then, we can skip computation of a shell quartet $(MN|PQ)$ if $\sqrt{s(MN)s(PQ)} < \tau$. There is also the associated concept of *significance*. A shell pair MN is significant if

$$s(MN) \geq \tau/m^*$$

where, for a basis set $\mathscr{B}$, m^* is defined as

$$m^* = \max_{M,N \in \mathscr{B}} s(MN).$$

It is also a common practice to compute integrals in larger batches called *atom quartets*. Recalling that an atom corresponds to a set of shells with the same center, an atom quartet $(I_{at}J_{at}|K_{at}L_{at})$, where $I_{at}, J_{at}, K_{at}, L_{at}$ are atoms, is defined as

$$(M_{at}N_{at}|P_{at}Q_{at}) = \{(MN|PQ) \text{ s.t. } M \in M_{at},\, N \in N_{at},\, P \in P_{at},\, Q \in Q_{at}\}$$

Screening can also be applied to atom quartets of integrals by an obvious extension of the procedure for shell quartets. Also, the concepts of significance and pair values extend to atom pairs trivially.

2.2 Challenges of Parallelizing Fock Matrix Construction

Fock matrix construction presents two main challenges for parallelization. The first is load imbalance arising from the irregularity of the independent tasks available in the computation. The irregularity is due to the structure of molecules and the screening procedure (see Section 2.1.4) used to reduce computational cost by neglecting the computation of insignificant electron repulsion integrals (ERIs). As mentioned in Section 2.1.3, the most computationally expensive part of Fock matrix construction is the computation of the ERIs. These can only be computed in batches of shell quartets, which may not be of the same size for different shells. Further, for large problems, many shell quartets are dropped by screening. In addition, even different shell quartets of ERIs with the same number of elements may take different times to compute. These factors make it hard to obtain an initial balanced partitioning of the computational volume of Fock matrix construction and lead to load imbalance problems.

The second challenge of parallelizing Fock matrix construction is the potentially high communication cost associated with the irregular data access pattern of Fock matrix construction. By way of explanation, we introduce the *Coulomb* and *exchange* matrices, J and K,

$$J_{ij} = \sum_{kl} D_{kl}(ij|kl)$$

$$K_{ij} = \sum_{kl} D_{kl}(ik|jl)$$

In terms of these, a Fock matrix can be expressed as

$$F = H^{core} + 2J - K \tag{5}$$

We defined J and K to emphasize the fact that they require *different* parts of the array of ERIs. Using Matlab-like indexing notation, J_{ij} requires $(ij|:,:)$, and K_{ij} requires $(i,:|j,:)$. Thus, if we want to exploit the symmetry of the ERIs and only compute unique ERIs, we need to phrase the construction of F in terms of the computation of the integrals $(ij|kl)$ rather than each F_{ij}.

In practice, as each shell quartet of integrals is computed, the corresponding blocks of F are updated. These are in different locations for the contributions arising through J, and K, resulting in a highly irregular pattern of accesses to D and F. For a shell quartet, $(MN|PQ)$, the blocks of F that need to be updated are F_{MN} and F_{PQ} for J, and F_{MP}, F_{NQ}, F_{MQ} and F_{NP} for K. Further, these updates require blocks D_{PQ}, D_{NQ}, D_{MN}, D_{MP}, D_{NP}, and D_{MQ} of D. Thus, for each shell quartet of integrals computed, six shell blocks of D need to be read and six shell blocks of F are updated. Note here that we use shell indices to denote the blocks of F and D corresponding to the basis function indices of the basis functions in the shells. This is common practice since indexing is arbitrary, and basis functions are usually indexed such that all the basis functions associated with a given shell are consecutively numbered. Thus, F_{MN} is nothing but the block,

$$F_{MN} = \{F_{ij} \text{ s.t. } i \in \text{shell } M, j \in \text{shell } N\}$$

Further, to tackle large problems we need to distribute the D and F matrices among the processors of a distributed system. The irregular pattern of access to D and F described above requires significant communication between processors and results in high communication cost.

Recent proposals to use GPUs and SIMD instructions for computing the ERIs [111, 75, 123, 81, 9, 118, 97, 98] will further make communication efficiency an important consideration in the parallel construction of Fock matrices. Thus, in order to achieve high performance for distributed parallel Fock matrix construction, it is desirable to have algorithms that reduce communication cost.

2.3 Limitations of Previous Work

The most well-known parallel algorithms for Fock matrix construction are from more than a decade ago [53, 44], which have been implemented in many computational chemistry packages including NWChem [112], GAMESS [102], ACESIII [73] and MPQC [61]. Here, we

take the implementation in NWChem as the example to demonstrate the limitations of the previous work.

The NWChem computational chemistry package distributes the matrices D and F in block row fashion among the available processes. The indices of the basis set are grouped by atoms, and if there are n_{atoms} atoms then process i owns the block rows of F and D corresponding to atom indices ranging from $(i \cdot n_{atoms}/p)$ to $((i+1) \cdot n_{atoms}/p) - 1$, where p is the total number of processes used. Note that in the above, we have assumed that p divides n_{atoms}.

Further, a task-based computational model is used in order to utilize the full permutational symmetry of the integrals. Each task is defined as the computation of 5 atom quartets of integrals, the communication of the parts of D corresponding to these blocks of integrals, and the updating of the corresponding parts of F. The choice of atom quartets as minimal computational units is made in order to increase data locality and reduce communication volume. Using Matlab-like notation, the task definition used in [53] is $(I_{at}J_{at}|K_{at}, L_{at} : L_{at}+4)$, where the $I_{at}, J_{at}, K_{at}, L_{at}$ are atom indices. This choice is a compromise between fine task granularity and low communication volume through data reuse [53, 44].

For load balancing, the tasks are dynamically scheduled on processes using a simple centralized dynamic scheduling algorithm. Processes extract tasks from a centralized task queue and execute them. Screening and symmetry consideration are incorporated into the task execution process, and only the unique significant shell quartets of integrals are computed.

This approach suffers from three problems. Firstly, the use of 5 atom quartets as a minimal unit of work does not allow for fine enough granularity when large numbers of processes are used, and as a result, load balancing suffers. It would seem at this point that a choice of a smaller unit of work could solve this problem, however the communication volume, and consequently the communication cost is also likely to increase if this is done.

Secondly, the task scheduling is completely dynamic, with no guarantee of which tasks get executed on which processes, so it is not possible to prefetch all the blocks of D required by a processes in a single step, before starting integral computation. Lastly the centralized dynamic scheduler is likely to become a bottleneck in cases when large numbers of cores are used.

The precise details of the algorithm are presented in Algorithm 2.

Algorithm 2: Parallel Fock Matrix Construction in NWChem.

```
1  On process p do
2  task ← GetTask() /*Access global task queue                    */
3  id ← 0
4  for Unique triplets I_at, J_at, K_at do
5      if (I_at J_at) is significant then
6          l_hi ← K_at
7          if K_at = I_at then
8              l_hi ← J_at
9          end
10         for l_lo ← 1 to l_hi stride 5 do
11             if id = task then
12                 for L_at ← l_lo to min(l_lo + 4, l_hi) do
13                     if (I_at J_at)(K_at L_at) > τ^2 then
14                         Fetch blocks of D
15                         Compute (I_at J_at | K_at L_at)
16                         Update blocks of F
17                     end
18                     id ← id + 1
19                 end
20                 task ← GetTask() /*Access global task queue  */
21             end
22         end
23     end
24 end
```

2.4 New Algorithm for Parallel Fock Matrix Construction

2.4.1 Overview

Our algorithm reduces communication costs while simultaneously tackling the problem of load balance by using an initial static task partitioning scheme along with a work-stealing

distributed dynamic scheduling algorithm. The reduced communication, along with the better scalability of work-stealing type scheduling algorithms [30], gives it better scalability than existing approaches.

We first specify an initial static task partitioning scheme that has reasonable load balance. The initial static partitioning allows us to know approximately on which processes tasks are likely to get executed, which in turn, allows us to perform all the communication for each process in a few steps. We also reorder shells in a basis set to increase overlap in the data that needs to be communicated by the tasks initially assigned to each process, which leads to a reduction in communication volume and hence communication cost.

2.4.2 Task Description

To describe the computation associated with tasks in our algorithm, we need the concept of what we call the *significant set* of a shell. Recalling the definition of significance from Section 2.1.4, we define the significant set of a shell M to be the set of all the shells N such that the pair MN is significant. More formally, this is the set

$$\Phi(M) = \{P \text{ s.t. } s(MP) \geq \tau/m^*\},$$

where τ and m^* retain their definitions from Section 2.1.4.

Also, we define the set $(M,:|N,:)$ corresponding to the shells M and N in the basis set, denoted by $\mathscr{B}$ to be,

$$(M,:|N,:) = \{(MP|NQ) \text{ s.t. } P \in \mathscr{B},\ Q \in \mathscr{B}\}$$

Now, a task is defined as the computation of the integrals $(M,:|N,:)$, and the updating of the corresponding blocks of the F matrix using the appropriate blocks of D. One can simply see that, after screening, $(M,:|N,:)$ contains $|\Phi(M)||\Phi(N)|$ significant shell quartets. That is,

$$(M,:|N,:) = \{(MP|NQ) \text{ s.t. } P \in \Phi(M),\ Q \in \Phi(N)\}$$

From Equation (5), the parts of D that need to be read, and the parts of the Fock matrix F that need to be updated are the shell blocks $(M,\Phi(M)),(N,\Phi(N)),(\Phi(M),\Phi(N))$. It can be seen that the six blocks of F and D associated with the computation of each shell quartet in a task are all contained within these parts of F and D. Further, for this definition of a task, the maximum number of tasks available for a problem with n_{shells} shells is n_{shells}^2.

Whether or not a pair of shells M and N is significant as defined in Section 2.1.4 is related to the distance between their centers, i.e., the distance between the atomic coordinates of the atoms that they are associated with. This, in a certain sense, implies that for a molecule whose atomic coordinates are distributed more or less uniformly in a contiguous region of space, the variation in $|\Phi(M)|$, for different shells M, should not be too large. This in turn implies that, for different shell pairs MN, the variation in $|\Phi(M)||\Phi(N)|$ should not be too large either. Thus, the amount of integral computation associated with different tasks should not vary widely.

2.4.3 Initial Static Partitioning

The tasks are initially equally distributed among processes. If we have a $p_{row} \times p_{col}$ rectangular virtual process grid, for a problem with n_{shells} shells in the basis set, tasks are initially assigned to processes in blocks of size $n_{br} \times n_{bc}$, where $n_{br} = n_{shells}/p_{row}$, and $n_{bc} = n_{shells}/p_{col}$. That is, the process p_{ij} is initially assigned the block of tasks corresponding to the computation of the set of shell quartets $(i \cdot n_{br} : (i+1) \cdot n_{br} - 1, : | j \cdot n_{bc} : (j+1) \cdot n_{bc} - 1, :)$. For simplicity, we have again assumed that p_{row} and p_{col} divide n_{shells}. From the comment at the end of the previous section, we expect that the time for integral computation for each task is approximately the same.

Having this initial static partitioning, each process can now prefetch the parts of D associated with the tasks assigned to it, and store them in a local buffer D_{local}. Also, a local buffer to hold the updates to F, F_{local}, is initialized before beginning the execution of integral computation. Subsequently, as shell quartets of integrals are computed, the process

uses data in D_{local} to update F_{local}.

On the surface it would seem that we do not consider the permutational symmetry of the ERIs as per Equation (3). However, with our task description we can enforce computation of only the unique integrals by computing a shell quartet only if certain relationships between its indices are satisfied. Consider a subroutine *SymmetryCheck*(M,N) for integers M, N, that returns *true* if either of the conditions, "$(M > N)$ *and* $(M+N)$" or "$(M \leq N)$ *and* $(M+N)$ is odd", is satisfied, and returns *false* otherwise. Computation of only unique shell quartets can be enforced using this on pairs of indices. Now we can give a complete specification of the operations performed in a task. This is presented in Algorithm 3. Note

Algorithm 3: *doTask*$(M,:|N,:)$

```
for Q ← 0 to n_shells − 1 do
    for P ← 0 to n_shells − 1 do
        if SymmetryCheck(M,N) and SymmetryCheck(M,P) and
        SymmetryCheck(N,Q) and (MN)(PQ) > τ then
            Compute (MP|NQ)
            Update blocks of F_local, using D_local
        end
    end
end
```

that in the above τ is the tolerance chosen for screening. Once computation is finished, the local F buffers can then be used to update the distributed F matrix.

2.4.4 Shell Reordering

The parts of D that need to be read, and F that need to be updated by a task that computes the integrals $(M,:|N,:)$ are given by the index sets $(M,\Phi(M)),(N,\Phi(N)),(\Phi(M),\Phi(N))$. As explained in the previous section, these parts corresponding to a block of tasks assigned to a process are prefetched.

Naturally, in order to reduce latency costs associated with this prefetching, we would like these regions of F and D to be as close in shape as possible to contiguous blocks. This would happen if $\Phi(M)$ and $\Phi(N)$ are such that the difference between the maximum and

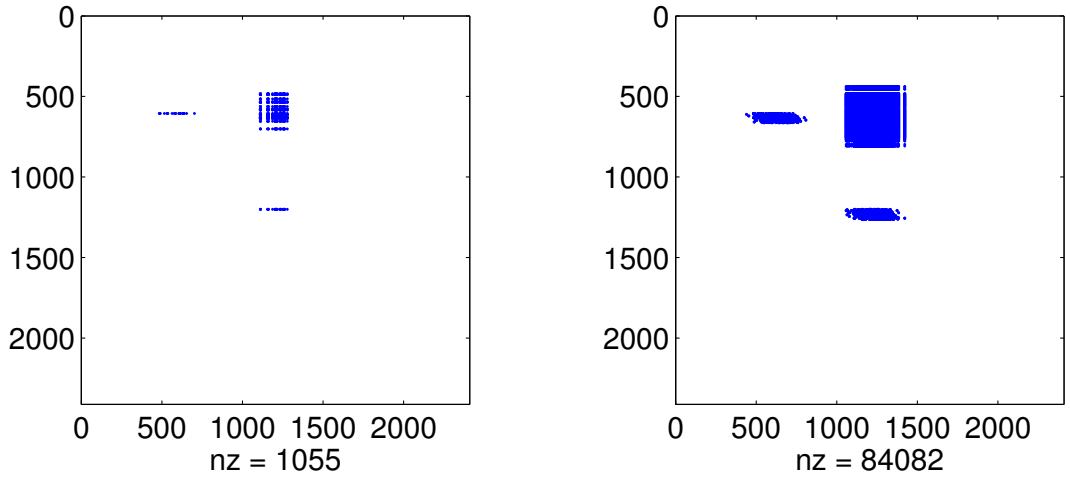
minimum shell indices in these sets is small. This can be achieved if pairs of shells that are significant have indices that are close together. Since the indexing of shells is arbitrary, and a shell pair is more likely to be significant if the distance between the centers of the pair is small, we could achieve this approximately by choosing an indexing scheme that numbers shells, whose centers are in close spatial proximity, similarly.

In our algorithm we utilize an initial shell ordering that does this to a certain extent. First, we define a three dimensional cubical region that contains the atomic coordinates of the molecule under consideration. This region is then divided into small cubical cells, which are indexed using a natural ordering. Then shells are ordered with those appearing in consecutively numbered cells being numbered consecutively, with the numbering within a cell being arbitrary. The basis function numbering is chosen so that basis functions within a shell are consecutively numbered, and the basis functions in two consecutively numbered shells form a contiguous list of integers.

This reordering has another very desirable consequence. In the initial partitioning scheme that we use, each process is assigned tasks that correspond to a block of shell pairs. As a result of this, once our shell ordering has been applied there is considerable overlap in the elements of F and D that need to be communicated by the tasks assigned to a process. This is illustrated by Figure 1 of which part (a) shows the parts of the density matrix D, and the number of elements, required by $(300,:|600,:)$, and (b) shows the parts required by the task block $(300:350,:|600:650,:)$ for the molecule $C_{100}H_{202}$ which has, with the cc-pVDZ basis set, 1206 shells and 2410 basis functions. The number of tasks in the latter block is 2500, however, the number of elements of D required is only about 80 times greater than that for the former single task.

2.4.5 Algorithm

Our algorithm assumes that both D and F are stored in distributed fashion, using a 2D blocked format. This is necessary when n_{shells} is large. The process p_{ij} in the process



(a) $(300,:|600,:)$ (b) $(300:350,:|600:650,:)$

Figure 1: Map of elements of D required by (a) $(300,:|600,:)$ and (b) $(300:350,:|600:650,:)$.

grid owns the shell blocks of F and D corresponding to the shell pair indices $(i \cdot n_{br} : (i+1) \cdot n_{br} - 1, j \cdot n_{bc} : (j+1) \cdot n_{bc} - 1)$, where the definitions of n_{br} and n_{bc} are the same as those in Section 2.4.3.

At the beginning of execution, each process populates a local task queue with the tasks assigned to it according to the static partition described in Section 2.4.3. It then prefetches all the blocks of D required by its tasks from the distributed D matrix and stores them in a contiguous buffer in local memory. A local buffer of appropriate size to hold updates to F for the tasks assigned to the process is also initialized. Subsequently, each task is extracted from the queue and executed. As explained in Section 2.4.2, this involves the computation of the shell quartets assigned to it, and the updating of the corresponding shell blocks of F using blocks of D. Because data has been prefetched, updates are performed to the local contiguous buffers. Once all the tasks are complete, the local F buffers are used to update the distributed F matrix. This is presented as Algorithm 2.

In practice, all communication operations are performed using the Global Arrays framework [84], which provides one-sided message passing operations, and is used to phrase communication in a manner similar to data access operations in shared memory.

Algorithm 4: FockBuild for process p_{ij}

```
Initialize task queue Q
Populate task queue with tasks (i·n_br : (i+1)·n_br − 1, : | j·n_bc : (j+1)·n_bc − 1, :)
Fetch and store required D blocks in D_local
Initialize F_local
while NotEmpty(Q) do
    (M, : |N, :) ← ExtractTask(Q)
    doTask(M, : |N, :)
end
Update F using F_local
```

2.4.6 Work-Stealing Scheduler

In spite of the fact that our initial partitioning scheme assigns blocks of tasks that have *similar* computational costs to process, it is not perfectly balanced. Dynamic load-balancing is required, which we achieve through the use of a simple work-stealing distributed dynamic scheduling algorithm [17]. This is implemented using Global Arrays.

When the task queue on a process becomes empty, it scans the processes in the process grid in a row-wise manner, starting from its row, until it encounters one with a non-empty task queue. Then it steals a block of tasks from this victim processor's task queue and adds it to its own queue, updating the victim's queue during this process. After this, it copies the local D buffer of the victim to its local memory and initializes a local buffer for updates to F corresponding to this, and updates these buffers during the execution of the stolen tasks. When a process steals from a new victim the current stolen F buffer is accumulated to F_{local} of the previous victim.

2.4.7 Performance Model and Analysis

In order to develop a model for the average running time of our algorithm, we have to make a few simplifying assumptions and define a few terms. The average time taken to compute an ERI is denoted by t_{int}. A square process grid is assumed with $p_{row} = p_{col} = \sqrt{p}$, p being the total number of processes used. It is also assumed that p divides n_{shells}. The average number of basis functions associated with a shell is denoted by A, and the average

number of shells in the set $\Phi(M)$, for a shell M, is denoted by B. The average number of elements in $\Phi(M) \cap \Phi(M+1)$ for shell M is q. We also assume that the average number of processors from which tasks are stolen by a given process, using the algorithm described in Section 2.4.6, is s. Lastly, the bandwidth of the communication network of the distributed system is taken to be β.

The computation cost of Fock matrix construction is the cost of computing the ERIs. The number of shell quartets in $(M,: |N,:)$ after screening is $|\Phi(M)||\Phi(N)|$, which has average value B^2. Thus, the number of shell quartets associated with the block of tasks assigned to each processor under the initial static partitioning scheme described in Section 2.4.3 is $n_{shells}^2 B^2/p$. Further, we only compute unique shell quartets of integrals. Thus, the number of shell quartets assigned to a process becomes $n_{shells}^2 B^2/8p$. Now, using A and t_{int}, the expression for average compute time is

$$T_{comp}(p) = \frac{t_{int} B^2 A^2 n_{shells}^2}{8p} \tag{6}$$

We recall from Section 2.4.2 that the task associated with the integrals $(M,: |N,:)$ needs to communicate shell blocks of F and D with shell pair indices $(M,\Phi(M))$, $(N,\Phi(M))$ and $(\Phi(M),\Phi(N))$. Each process p_{ij} under the initial static partitioning owns the task block corresponding to the integrals $(i \cdot n_b : (i+1) \cdot n_b - 1,: |j \cdot n_b : (j+1) \cdot n_b - 1,:)$, where $n_b = n_{shells}/\sqrt{p}$. Thus, the average communication volume for a process arising from the need to communicate blocks of F and D, corresponding to the blocks $(M,\Phi(M))$ and $(N,\Phi(N))$, for shells M and N in $(i \cdot n_b : (i+1) \cdot n_b - 1)$ and $(j \cdot n_b : (j+1) \cdot n_b - 1)$, respectively, is

$$v_1(p) = 4A^2 B n_{shells}^2/p \tag{7}$$

The communication volume associated with the blocks $(\Phi(M),\Phi(N))$ is a little more tricky to obtain since we have to take into consideration the overlap in these sets for the tasks associated with a process, as explained in Section 2.4.4. If $\Phi(M)$ and $\Phi(M+1)$ have q elements in common, and the average value of $|\Phi(M)|$ is B, then the average number of elements in $\Phi(M+1) \cup \Phi(M) - \Phi(M) \cap \Phi(M+1)$ is $2(B-q)$. Thus, $|\Phi(M) \cup \Phi(M+1)|$

should be $q+2(B-q)$. This can be extended to $n_{shells}/\sqrt{p}$ shells, giving the expression $(q+(n_{shells}/\sqrt{p})(B-q))$ for the average number of elements in the union of the Φ sets corresponding to these shells. Thus, the average communication volume arising from these sets for a process is

$$v_2(p) = 2\left(\frac{n_{shells}}{\sqrt{p}}(B-q)+q\right)^2 A^2 \tag{8}$$

The average communication volume, including the communication for D and F buffers for steals, is

$$V(p) = (1+s)\,(v_1(p)+v_2(p)) \tag{9}$$

Now, an expression for the communication time is

$$T_{comm}(p) = \frac{1}{\beta}V(p) \tag{10}$$

With equations (6) and (10), we are now in a position to arrive at an expression for the *efficiency* of the new parallel algorithm. Efficiency is given by

$$E(p) = \frac{T^*}{pT(p)},$$

where T^* is the *running* time of the fastest sequential algorithm for solving the same problem as the parallel algorithm. In our case, since we utilize screening, and only compute unique ERIs, we make the assumption that $T^* = T_{comp}(1)$. Also, $pT_{comp}(p) = T_{comp}(1)$, and we assume no overlap in computation and communication, so $T(p) = T_{comp}(p)+T_{comm}(p)$, yielding

$$E(p) = \frac{1}{1+T_{comm}(p)/T_{comp}(p)}.$$

Thus efficiency depends on the ratio $T_{comm}(p)/T_{comp}(p)$, which we denote by $L(p)$. Using equations (7), (8), (9), (10) and (6) and some trivial algebraic manipulations we get

$$L(p) = \frac{16(1+s)}{t_{int}B^2\beta}\left(\left((B-q)+q\frac{\sqrt{p}}{n_{shells}}\right)^2+2B\right). \tag{11}$$

Expression (11) tells us several things, the first being the *isoefficiency function* of our algorithm, which is defined as the rate at which the problem size must vary in terms of the

number of processes, in order to keep efficiency constant. Efficiency is constant in our case if L is constant and L is constant if $\sqrt{p}/n_{shells}$ is constant, assuming that s does not vary with p. This gives us an isoefficiency function of $n_{shells} = O(\sqrt{p})$. Hence, in order for us to have constant efficiency, the problem size, specified in terms of the number of shells, must grow at least as fast as $\sqrt{p}$.

In the above analysis we have assumed s to be a constant. This is expected to be true if both p and n_{shells} (and hence the amount of computational work) are increasing.

Equation (11) also gives us some other qualitative information. Substituting $p = n_{shells}^2$, the maximum available parallelism, we obtain

$$L(n_{shells}^2) = \frac{16(1+s)}{t_{int}\beta}\left(1+\frac{2}{B}\right) \tag{12}$$

Now, with increasing number of processes, the algorithm will only reach a point at which communication starts to dominate if this is greater than 1. This is likely to happen sooner as t_{int} goes down, with improvements in integral calculation algorithms and technology. Further, for highly heterogeneous problems with many widely varying atom types that are irregularly distributed in space, we expect that our initial task partitioning will be less balanced, implying that the number s is likely to go up, making communication dominate sooner.

The presence of the term $2/B$ tells us about the effect that the structure of the molecule has on the running time. B is the average value of $|\Phi(M)|$, which is the number of shells that have a significant interaction with M, and this number is expected to be very large for a molecule, that has atoms centered at points that are densely distributed in three dimensional space. This larger value indicates, from expression (12), that computation dominates for such a problem, as expected.

Expression (12) can be used to determine how much smaller t_{int} needs to be for there to exist a point at which communication costs start to dominate. Consider the molecule $C_{96}H_{24}$. With the cc-pVDZ basis set it was observed that, using 3888 cores on a test machine (described in Section 2.6), the average value for s for our Fock matrix construction

algorithm was 3.8. For simplicity we assume that, B is large so that $2/B \approx 0$ for this problem. Also, the bandwidth of the interconnect of the test machine was 5 GB/s. Using $t_{int} = 4.76\mu s$ from Table 6 in Section 2.6 and expression (12) we can arrive at the conclusion that integral computation has to be approximately 50 times faster for there to exist a point at which communication starts to dominate. This is supported by the results in Figure 2, which indicate that this case is still heavily dominated by computation with 3888 cores. In contrast to this, for NWChem's algorithm described in Section 2.3, the parallel overhead time, of which communication cost is a component, actually becomes greater than the computation time at $p \approx 3000$ (refer to Figure 2).

In all the analysis in this section, we have made no mention of the latency costs associated with communication. We do this for simplicity. All that can be said is that the latency costs will add to the communication time, increasing $L(p)$ and reducing the critical number of processes at which communication costs surpass computation costs.

2.5 Heterogeneous Fock Matrix Construction

In this section, we will describe a hybrid implementation of our new algorithm for Fock matrix construction using both CPUs and Intel Xeon Phi. The usual challenges are 1) load-balancing the calculations between CPUs and Intel Xeon Phi coprocessors, and 2) reducing data transfer overhead across the PCI-e bus.

We address the first challenge by using work-stealing dynamic scheduling between CPUs and coprocessors. Due to the variability in the computational cost of calculating integrals, static partitioning has poor load balancing performance for integral calculations. On the other hand, work-sharing dynamic scheduling between CPU and Xeon Phi has high communication overhead. Using a work-stealing scheduler, we are able to address the two problems simultaneously. Specifically, a dedicated CPU core is used to schedule integral calculations for all the Xeon Phi coprocessors on a node. Initially, integrals are divided into small tasks and statically assigned to all the computing resources on the node. When

a Xeon Phi card finishes computing its own tasks, the dedicated CPU core will steal tasks from a busy resource's task queue.

Our heterogeneous implementation computes both integrals and the Fock matrix on Xeon Phi, which only requires transferring the Fock and the density matrices. Compared to the simpler task of only computing integrals on Xeon Phi, this significantly reduces the communication requirements between the coprocessor and the host since the total size of the Fock and density matrix blocks is much smaller than the number of integrals computed. Essentially, a reduction operation is performed to compute the Fock matrix, so communicating blocks of the Fock matrix is much more efficient than communicating integrals.

However, computing the Fock matrix on Intel Xeon Phi requires storing both the Fock and density matrices on the Xeon Phi card. It is not feasible to store The entire the limited storage on an Intel Xeon Phi card can limit problem sizes. To address this problem, we use a similar partitioning of integrals as what is used in the distributed workload partitioning. For each partitioning of integrals, the required Fock matrix and density matrix blocks can be pre-computed, and the blocks are usually small enough to put on Xeon Phi.

To compute the Fock matrix in parallel, a straightforward approach is to store multiple copies of Fock matrices (more precisely, blocks of the Fock matrix being constructed). Each thread updates one copy independently of other threads, and a reduce operation is performed at the end. While this approach works very well for CPUs, we usually do not have enough space to store 224 copies (1 copy for each thread) of Fock matrices on Xeon Phi. The other approach to this problem is to use atomic operations. However, atomic operations have relatively low performance on Xeon Phi due to the large number of cores.

To address this problem, we combine the above two approaches. First, it is unnecessary to store one Fock matrix copy for each Xeon Phi thread. Four threads within a core can share one copy, as atomic operations within a core have low overhead. Second, for each task, there are six blocks of F that need to be updated (due to symmetries in the ERI tensor). We make the observation that not all of these blocks are of the same size (due to different

types of shells). Instead of storing copies of all of these blocks, we only store copies for the smaller blocks, and use atomic operations for single copies of the larger blocks. We found experimentally that this approach introduces less than 5% overhead due to atomic operations, while the memory requirement remains manageable.

2.6 Experimental Results

2.6.1 Experimental Setup

Tests were conducted on the Lonestar supercomputer[1] located at the Texas Advanced Computing Center. Each node of Lonestar is composed of two Intel Westmere X5680 processor (12 cores each at 3.33 GHz). The nodes of Lonestar are connected by an InfiniBand Mellanox switch with a bandwidth of 5 GB/s. The normal queue was used which allows a maximum number of 4104 cores (342 nodes) to be requested. Timings for heterogenous Fock matrix construction were collected using a single node server composed of two Intel Westmere X5680 and two Intel Xeon Phi coprocessors (61 cores; SE10P).

The implementation of our algorithm, which we refer to as GTFock, was compared to the distributed Fock matrix construction algorithm implemented in NWChem version 6.3 [44, 53]. Our implementation uses Global Arrays [84] version 5.2.2, which is the same version used by NWChem 6.3. We also use the MPICH2 version 2.1.6 implementation of MPI-2. For ERI computation, the ERD integrals package [43] is used. This is distributed as part of the ACES III computational chemistry package [73]. Intel compilers ifort version 11.1 and icc version 11.1 were used to compile both NWChem and GTFock, and both were linked against the Intel MKL version 11.1 libraries.

Our implementation uses OpenMP multithreading to parallelize the computations associated with a task, given in Algorithm 3. Consequently, we ran GTFock with one MPI process per node. NWChem does not use multithreading, and one MPI process per core was used in this case.

[1] https://www.tacc.utexas.edu/resources/hpc/lonestar

We used four test molecules with the Dunning cc-pVDZ basis set [34]. The test cases along with their properties are presented in Table 2. The first two molecules have a 2D planar structure similar to the carbon allotrope, graphene. The latter two are linear alkanes, which have a 1D chain-like structure. A screening tolerance of $\tau = 10^{-10}$ was used for all tests, for both our implementation and for NWChem, and for a fair comparison, optimizations related to symmetries in molecular geometry were disabled in NWChem.

Table 2: Test molecules.

Molecule	Atoms	Shells	Functions	Unique Shell Quartets
$C_{96}H_{24}$	120	648	1464	1.19×10^9
$C_{150}H_{30}$	180	990	2250	3.12×10^9
$C_{100}H_{202}$	302	1206	2410	1.68×10^9
$C_{144}H_{290}$	434	1734	3466	3.52×10^9

2.6.2 Performance of Heterogeneous Fock Matrix Construction

On each compute node, one CPU thread dynamically offloads work to all the Intel Xeon Phi cards. The remaining CPU threads use dynamic scheduling to distribute work among them. We collected timings for the heterogeneous Fock matrix construction on a single node and calculated the offload efficiency for each test molecule. The results are shown in Table 3.

The offload efficiency for dual WSM and dual Intel Xeon Phi is also shown in Table 3. Offload efficiency is defined as the ratio of two speedups: the actual speedup vs. theoretical speedup. The actual speedup is the speedup of dual WSM with dual Phi over single WSM. The theoretical speedup is the speedup if the dual WSM and dual Phi ran independently with no offload overheads. More precisely, if one Phi behaves like F WSM processors, then with two WSM processors and two Phi coprocessors, the theoretical speedup is $(2+2F)$. The quantity F may be measured as the ratio of the time consumed for one WSM processor vs. the time consumed by one Phi coprocessor for the same workload. Table 3 shows that

this offload efficiency is high, indicating little overhead due to offloading and dynamic scheduling to the four processing components in this test.

Table 3: Speedup compared to single socket Westmere (WSM) processor.

Molecule	single WSM	single Phi	dual WSM	dual WSM with dual Phi	Offload efficiency
$C_{96}H_{24}$	1	2.11	1.98	5.78	0.929
$C_{150}H_{30}$	1	2.17	1.99	5.92	0.934
$C_{100}H_{202}$	1	2.27	2.00	6.18	0.945
$C_{144}H_{290}$	1	2.35	2.00	6.37	0.951

2.6.3 Performance of Distributed Fock Matrix Construction

Table 4 compares the running time for Fock matrix construction for NWChem and GTFock for the test cases just described. Although NWChem is faster for smaller core counts, GTFock is faster for larger core counts. Table 5 shows the corresponding speedups and shows that GTFock has better scalability than NWChem up to 3888 cores.

Table 4: Fock matrix construction time (in seconds) for GTFock and NWChem on four test cases. Although NWChem is faster for smaller core counts, GTFock is faster for larger core counts.

Cores	$C_{96}H_{24}$		$C_{150}H_{30}$		$C_{100}H_{202}$		$C_{144}H_{290}$	
	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem
12	673.07	649.00	1765.92	1710.00	619.15	406.00	1290.58	845.00
108	75.17	75.00	196.95	198.00	68.32	48.00	140.14	97.00
192	42.53	42.00	111.02	115.00	38.58	27.00	78.92	57.00
768	10.78	12.00	28.03	29.00	9.72	7.40	19.91	15.00
1728	4.93	5.30	12.57	13.00	4.37	4.80	9.03	7.30
3072	2.91	4.10	7.21	8.50	2.50	5.10	5.11	5.30
3888	2.32	4.50	5.80	6.70	2.02	5.80	4.06	9.00

We used the NWChem running time on a single node to compute the speedup for both NWChem and GTFock, since NWChem is faster on a single node. NWChem's better single-node performance is most likely due to its better use of *primitive pre-screening* [43]

Table 5: Speedup in Fock matrix construction for GTFock and NWChem on four test cases, using the data in the previous table. Speedup for both GTFock and NWChem is computed using the fastest 12-core running time, which is from NWChem. GTFock has better speedup at 3888 cores.

Cores	$C_{96}H_{24}$		$C_{150}H_{30}$		$C_{100}H_{202}$		$C_{144}H_{290}$	
	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem
12	11.57	12.00	11.62	12.00	7.87	12.00	7.86	12.00
108	103.60	103.84	104.19	103.64	71.31	101.50	72.36	104.54
192	183.10	185.43	184.84	178.43	126.27	180.44	128.48	177.89
768	722.72	649.00	732.15	707.59	501.44	658.38	509.22	676.00
1728	1581.00	1469.43	1631.94	1578.46	1114.87	1015.00	1122.43	1389.04
3072	2678.13	1899.51	2847.23	2414.12	1949.58	955.29	1983.57	1913.21
3888	3354.01	1730.67	3540.98	3062.69	2415.47	840.00	2498.77	1126.67

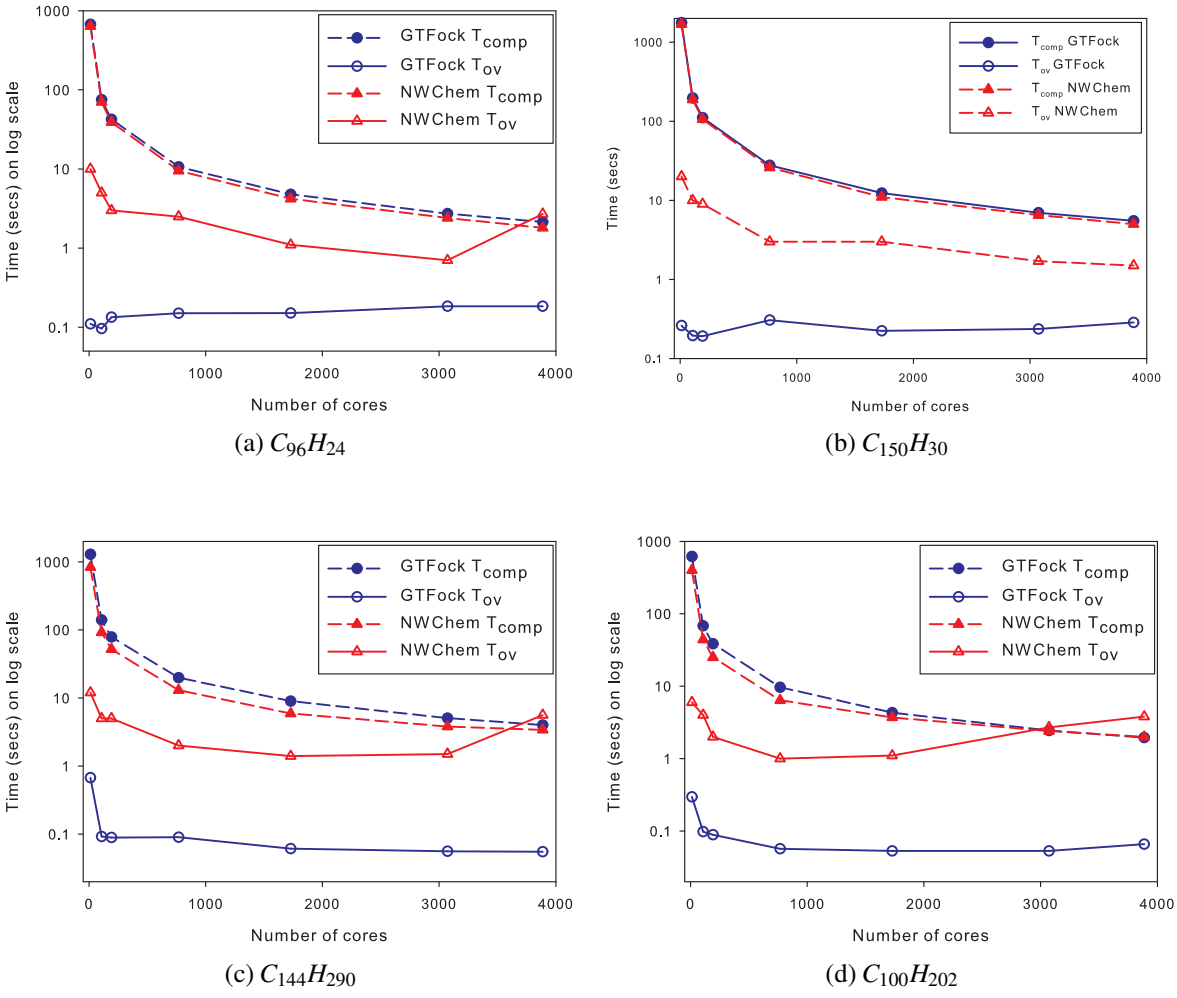
to avoid computation of negligible contributions to integrals. The results in Table 6 compare the performance of the integral packages of both implementations on a machine with similar characteristics as one node of our test machine, for two molecules that are representative, structurally, of the molecules that we used to test Fock matrix construction. The difference for the alkane case $C_{10}H_{22}$ is accentuated because primitive pre-screening is likely to be more effective due to the spatial distribution of atoms of this molecule.

Table 6: Average time, t_{int}, for computing each ERI for GTFock (using the ERD library) and NWChem.

Mol.	Atoms/Shells/Funcs	t_{int} GTFock	t_{int} NWChem
$C_{24}H_{12}$	36/180/396	4.759 μs	3.842 μs
$C_{10}H_{22}$	32/126/250	3.060 μs	2.400 μs

To understand the above Fock matrix construction timing results, for each of GTFock and NWChem, we measured the average time per process, T_{fock}, and the average computation-only time per process, T_{comp}. We assume that the average parallel overhead is $T_{ov} = T_{fock} - T_{comp}$. Figure 2 plots these quantities for different numbers of cores for our four test molecules.

We note that, for all cases, the computation time for GTFock is comparable to that of NWChem, with computation in NWChem being slightly faster for reasons explained in the



(a) $C_{96}H_{24}$ (b) $C_{150}H_{30}$

(c) $C_{144}H_{290}$ (d) $C_{100}H_{202}$

Figure 2: Comparison of average computation time T_{comp} and average parallel overhead time T_{ov} of Fock matrix construction for NWChem and GTFock. The computation times for NWChem and GTFock are comparable, but GTFock has much lower parallel overhead.

previous paragraph. However, in every case, the parallel overhead for GTFock is almost an order of magnitude lower than that for NWChem. For cases in Figures 2(a), (c), and (d), the overhead time for NWChem actually becomes comparable or greater than the average computation time for larger numbers of cores. This is due to the fact that there is relatively less work for these cases. For the alkane cases, there is less computational work because the molecules have a linear structure and there are many more shell quartets of integrals neglected due to screening. For the smaller graphene case, this is due to the fact that the amount of available computation is less. The increased proportion of parallel overhead

is the reason for the poorer scalability of NWChem on these test cases, shown earlier in Table 4 and Table 5.

2.6.4 Analysis of Parallel Overhead

The parallel overhead time T_{ov}, illustrated in Figure 2, has three main sources: communication cost, load imbalance, and scheduler overhead from atomic accesses to task queues. We provide evidence to show the reduced communication cost of our algorithm versus that of NWChem. The cost of communication on a distributed system is composed of a latency term and a bandwidth term.

The number of calls to communication functions in Global Arrays and the number of bytes transferred provide qualitative indicators of latency and bandwidth, respectively. We measured these quantities for NWChem and GTFock. These results are presented in Table 7 and Table 8. We see that our implementation has lower volumes and numbers of calls for all the cases, indicating a lower communication cost, and explaining the reduced parallel overhead. It should be noted here that the volumes measured are total communication volumes, including local transfers. This was done in order to have a fair comparison between NWChem and GTFock since, as mentioned previously, the number MPI processes per core was different for each of them.

Scheduler overhead for NWChem can be inferred indirectly from the number of accesses to the task queue of its centralized dynamic scheduler. The number of such accesses for the case $C_{100}H_{202}$ with 3888 cores is 330091. Each of these operations must be atomic and it is expected that a serialization cost is incurred by them. In comparison, our work-stealing scheduler only needs the execution of 349 atomic operations on each of the task queues of the nodes. The serialization due to this is likely to be much less.

2.6.5 Load Balance Results

Our algorithm uses a work-stealing distributed dynamic scheduling algorithm to tackle the problem of load imbalance in Fock matrix construction. In this section we present

Table 7: Average Global Arrays communication volume (MB) per MPI process for GTFock and NWChem.

Cores	$C_{96}H_{24}$		$C_{150}H_{30}$		$C_{100}H_{202}$		$C_{144}H_{290}$	
	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem
12	15.00	291.30	35.37	1020.79	40.61	457.16	84.08	1046.94
48	8.70	50.24	17.67	143.29	10.01	66.26	3.34	136.11
192	11.77	18.27	16.80	83.41	7.40	40.13	2.70	79.11
768	10.44	16.58	18.20	38.21	4.05	18.48	1.84	30.46
1728	7.63	11.36	12.87	22.03	3.56	14.87	3.90	17.95
3072	7.36	7.40	11.40	15.51	2.43	10.48	3.11	15.72
3888	6.24	8.23	9.94	16.37	2.38	8.26	2.78	14.44

Table 8: Average number of calls to Global Arrays communication functions per MPI process for GTFock and NWChem.

Cores	$C_{96}H_{24}$		$C_{150}H_{30}$		$C_{100}H_{202}$		$C_{144}H_{290}$	
	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem
12	11	3,204	11	10,758	11	9,590	11	19,898
48	33	678	71	1,401	28	1,379	29	2,587
192	59	610	81	822	28	955	33	1,510
768	65	602	103	840	28	988	38	1,023
1,728	127	711	123	758	32	1,386	35	950
3,072	129	533	148	637	30	1,253	32	1,189
3,888	147	1,091	170	1,008	30	989	31	1,357

Table 9: Load balance ratio $l = T_{fock,max}/T_{fock,avg}$ for four test molecules. A value of 1.000 indicates perfect load balance.

Cores	$C_{96}H_{24}$	$C_{150}H_{30}$	$C_{100}H_{202}$	$C_{144}H_{290}$
12	1.000	1.000	1.000	1.000
108	1.021	1.011	1.015	1.023
192	1.031	1.019	1.024	1.022
768	1.026	1.031	1.021	1.027
1728	1.042	1.037	1.025	1.021
3072	1.039	1.035	1.032	1.023
3888	1.065	1.035	1.030	1.021

experimental results that demonstrate the effectiveness of this approach to load balancing on our chosen test molecules.

Load balance can be expressed as the ratio of the longest time taken to complete the

operations of Fock matrix construction for any process, to the average time. This is the ratio $l = T_{fock,max}/T_{fock,avg}$. A computation is perfectly load balanced if this ratio is exactly 1.

The ratios l for different numbers of processes, for the test molecules considered, are presented in Table 9. The results indicate that in all the cases the computation is very well balanced, and that our load balancing approach is effective.

2.7 Summary

Fock matrix construction is a fundamental kernel in quantum chemistry. Existing algorithms and software, however, may fail to scale for large numbers of cores of a distributed machine, particularly in the simulation of moderately-sized molecules. In this chapter, we presented a new scalable algorithm for Fock matrix construction for the HF algorithm. We addressed two issues: load balance and the reduction of communication costs. Load balance was addressed by using fine-grained tasks, an initial static task partitioning, and a work-stealing dynamic scheduler. The initial static task partitioning, augmented by a reordering scheme, promoted data reuse and reduced communication costs. Our algorithm has measurably lower parallel overhead and shows better scalability than the algorithm used in NWChem for moderately-sized problems chosen to accentuate scalability issues.

We expect that the technology for computing ERIs will improve, and efficient full-fledged implementations of ERI algorithms on GPUs will reduce computation time to a fraction of its present cost. This will, in turn, increase the significance of new algorithms such as ours that reduce parallel overhead.

Chapter III

HARTREE-FOCK CALCULATIONS ON LARGE-SCALE DISTRIBUTED SYSTEMS

The Hartree-Fock (HF) method is composed of two main computational components: Fock matrix construction, and computation of a density matrix. We have presented a new parallelization for Fock matrix construction, which shows better scalability than NWChem. In this chapter we will address the problem of computing density matrices and present an efficient implementation of HF calculations on large-scale distributed systems.

3.1 Current State-of-the-Art

There exist many computational chemistry packages that implement HF for distributed computation, including NWChem [112], GAMESS [102], ACESIII [73] and MPQC [61]. The largest HF calculation of which we are aware is for a TiO_2 supercell with a single Fe dopant, consisting of 1,920 atoms and 40,320 basis functions. The calculation used the CRYSTAL program, running on 8,192 cores of a CRAY XT6 system, and required approximately 600 seconds per SCF iteration [22]. For proteins, the largest HF calculation to our knowledge, also performed by CRYSTAL, is of the crambin protein with 1,284 atoms and 12,354 basis functions [10]. CRYSTAL used the PDSYEVD divide-and-conquer eigensolver in ScaLAPACK for the density matrix computation. For one SCF iteration on 1,024 processors of an IBM POWER5 cluster required more than 200 seconds. For this size problem, the eigensolver required a small fraction of the time, but did appear not scale past 256 processors.

For large numbers of nodes and cores, scaling up the density matrix computation is a challenge. Benchmarks on NWChem (using Gaussian basis set DFT) for a C_{240} system

with 3,600 basis functions shows only a 10× speedup as the number of threads is increased from 32 to 4,096 [1]. Our own tests on Stampede for problems with approximately 10,000 basis functions show that eigensolver approaches do not scale past 100 nodes. The eigensolver can thus limit overall scalability.

We note that HF is the most basic approximation for solving the Schrödinger equation. More accurate methods take electron correlation into account, for example, many-body perturbation theory and coupled-cluster theory [12]. Almost all high performance computing research in quantum chemistry focuses on parallelizing these electron correlation methods. The coupled-cluster method CCSD(T), for example, scales as $O(n^7)$ and is based on tensor contractions which can be implemented using flop-rich matrix-matrix multiply kernels. Some of the highest flop rate quantum chemistry calculations have been performed using CCSD(T). NWChem achieved a flop rate of 487 Tflops/s on 96,000 processors of the Jaguar supercomputer using this method for a problem with 1,020 basis functions [8]. More recently, in 2013, the Cyclops tensor framework was used for a CCSD calculation on 4,096 nodes of BlueGene/Q for a problem in excess of 800 basis functions [106]. Although flop rates were not reported for problems of this size, excellent parallel efficiency was observed. Importantly, symmetry was exploited by using Cyclops, giving CCSD run times approximately half of that of NWChem. Also, other tools have been developed for high performance tensor contractions including the Tensor Contraction Engine [13]. In contrast, our work addresses the HF method, which is not based on dgemm and which is much more challenging to parallelize efficiently and scale than coupled-cluster methods [53, 44].

3.2 Improving Parallel Scalability of Fock Matrix Construction

In Chapter 2 we have presented a new scalable parallel algorithm for Fock matrix construction. While that new algorithm scales very well on Lonestar up to 3,888 cores (324 nodes), we observed that the performance was relatively poor on larger distributed systems. This is mainly due to the overhead of the work-stealing dynamic scheduling.

To address the problem, we first improved the initial static partitioning, so that work stealing is needed less often. To describe the improved static partitioning, recall that the significant set of a shell is defined as

$$\Phi(M) = \{P \text{ s.t. } s(MP) \geq \tau/m^*\},$$

Let $\eta(M)$ to be the number of elements in $\Phi(M)$. An upper bound on the number of shell quartets in $(M,:|P,:)$ that survive screening is $\eta(M)\eta(P)$. For p MPI processes, we group the shells into $\sqrt{p}$ groups so that each group G_i has the approximately equal values of η^*, which is defined as

$$\eta^* = \sum_{M \in G_i} \eta(M)$$

The estimated number of shell quartets in each partition is thus $\eta^* \times \eta^*$, which is balanced.

Second, we implemented a more intelligent work stealing scheme, one that is better balanced and has fewer failed steals. Specifically, we group nodes and maintain a global array to indicate which groups still have tasks. When one process finishes its own work, it will first read the global array to select the victim group. This will reduce the number of failed steals and can significantly reduce overhead. When a victim group is determined, the thief process randomly selects a victim process from the victim group and each time steals half of the tasks from the victim process. This random victim selection policy has better load balancing performance and lower stealing overhead than the row-wise scan policy used in our previous work. Algorithm 5 shows the intelligent work stealing scheme.

3.3 Optimization of Integral Calculations

High performance of the ERI calculations (see Section 2.1.3) on each node is essential for high performance of the overall distributed HF code. We use the ERD integral library [43] and optimized it for the Intel Ivy Bridge architecture, although we expect these optimizations to also benefit other modern CPUs. The ERD library uses the Rys quadrature method [35, 100] which has low memory requirements and is efficient for high angular

Algorithm 5: Hierarchical work-stealing dynamic scheduling on a $N_p \times N_p$ virtual process grid. The processes are grouped by row, that is, the process row i forms the process group PG_i. The global array W indicates which groups still have tasks. Initially all entries of W are set to one. $W_i = 1$ means PG_i still has remaining tasks.

```
1  On a process in PG_k do
2  repeat
3  |  Select a victim process group PG_l that is closest to PG_k and W_l = 1
4  |  C ← ∅
5  |  repeat
6  |  |  Randomly select a process p from PG_l
7  |  |  if p ∉ C then
8  |  |  |  repeat
9  |  |  |  |  steal half of the tasks from p
10 |  |  |  until the task queue of p is empty
11 |  |  |  C ← C ∪ p
12 |  |  end
13 |  until |C| = N_p
14 |  W_l ← 0
15 until all entries of W are equal to zero
```

momentum basis functions compared to other methods. It contains nearly 30,000 lines of Fortran code that provides the functionality to compute ERIs.

The computation of a shell quartet of ERIs requires several steps, including computation of Rys quadrature roots and weights, computation of intermediate quantities called 2D integrals using recurrence relations, computation of the constants used in the recurrence relations (which depend on the parameters of the basis functions in each shell quartet), and computation of each ERI from the 2D integrals. Reuse of the 2D integrals in a shell quartet is what requires ERI calculations to be performed at least one shell quartet at a time.

Due to the many steps involved in computing ERIs, there is no one single kernel that consumes the bulk of the time. Each step consumes a small percentage of the total time. We thus had a considerably large optimization effort, encompassing nearly 30,000 lines of Fortran code.

Loops were restructured so that they could be vectorized and exploit wide SIMD units. For example, the "vertical" recurrence relations can be computed in parallel for different

quadrature roots and different exponents. To increase developer productivity, we relied on programmer-assisted compiler auto-vectorization for several parts of the ERI code. We aligned most arrays on the width of SIMD registers and padded them to be a multiple of the SIMD register size. We annotated pointer arguments with the *restrict* keyword, and used Intel compiler-specific intrinsics *__assume_aligned* and *__assume* to convey information about array alignment and restrictions on variable values. We also employed *#pragma simd* to force loop vectorization. In addition, we explicitly vectorized several hot-spots using intrinsics where the compiler lacked high-level information to generate optimal code. Wherever possible and beneficial, we tried to merge multiple loop levels to get a larger number of loop iterations, thereby, achieving better SIMD efficiency. In almost every function, we also had to work around branching inside loops, in order to fully vectorize the code. We note, however, that in ERI calculations, there are many scalar operations and the loops generally have a relatively small number of iterations, limiting the performance gains on SIMD hardware.

In the process of converting the baseline Fortran code to C99, we found several low-level optimizations to be beneficial. First, instead of using globally allocated scratch arrays, we used local on-stack variable-length arrays provided by the C99 standard. Second, we converted signed 32-bit indices to unsigned 32-bit indices because, on x86-64, the processor needs to extend a signed 32-bit to 64 bits to use it as an index for a memory load or store. For unsigned indices, an extension instruction is not necessary because any operation on the low 32 bits implicitly zeroes the high part. Third, we found register pressure to be a problem. The ERD code was originally developed for the PowerPC architecture, which has 32 general purpose and 32 floating-point registers. On x86-64 we have only 16 general-purpose and 16 floating-point/SIMD registers. In our optimized code we revised interfaces to reduce the number of function parameters, and thus lowered the register pressure.

We observed that 30% of ERD computation time is devoted to primitive screening. This operation computes an upper bound on the size of the integrals in a shell quartet. If

this upper bound is above a threshold, then the shell quartet is not computed. The bound, however, requires division and square root operations, which are not pipelined on the CPU. Computation of this bound was rearranged to avoid these operations. Furthermore, the bound also requires computing the Boys function. The zero-order Boys function is defined as

$$F_0(x) = \int_0^1 \exp\left(-t^2 x\right) dt = \begin{cases} \frac{\sqrt{\pi}}{2} \frac{\operatorname{erf}\sqrt{x}}{\sqrt{x}} & \text{if } x > 0 \\ 1 & \text{if } x = 0. \end{cases}$$

The original ERD approximates the Boys function via a set of Taylor expansions on a grid of points. Such an approximation, however, is suboptimal for the following reasons. First, although Taylor expansions give good approximations in the vicinity of tabulated points, they are much less accurate away from tabulated points. Second, fetching coefficients at tabulated points in vectorized code requires a gather operation, which is lacking in Ivy Bridge processors and is expensive, albeit supported, on Intel Xeon Phi coprocessors. We derived a more efficient way to compute the Boys function, based on Chebyshev polynomials, which can minimize the maximum error over an interval. The main idea is to manipulate the bound so that we need to approximate $(\operatorname{erf}\sqrt{x})^2$. This can be well approximated by a 5th-degree Chebyshev polynomial approximation.

3.4 Computation of the Density Matrix

A major component of SCF iterations is the computation of the density matrix, D, which typically involves computing the eigenvectors of the Fock matrix, F. (For simplicity, in this section, we ignore the basis transformation, X.) Although massive parallel resources can be used to compute F of moderate size, the same resources cannot be efficiently employed to compute the eigenvectors of F, due to the relatively small workload and lack of large amounts of parallelism in this computation.

Recall D is computed as

$$D = C_{occ} C_{occ}^T$$

where C_{occ} is the matrix formed by the lowest energy eigenvectors of F. For small problems, computing all the eigenvectors of F is reasonable. For large problems, much research has been devoted to develop iterative methods for computing a large number of eigenvectors corresponding to the lowest eigenvalues. The eigenproblem can be a scalability bottleneck, due to this need to solve a less scalable problem on the critical path of an SCF iteration.

Our solution is to use a "diagonalization-free" method that avoids solving an eigenvalue problem and computes D directly from F (rather than through C_{occ}, which can be computed if necessary at the end of the SCF iterations). The method, in its most basic form, is known as McWeeny purification [77]. The algorithm is based on matrix-matrix multiplies, starting with an appropriate D_0,

$$D_{k+1} = 3D_k^2 - 2D_k^3$$

and thus it can be very efficient on modern processors, including in distributed environments. This method scales much better than computing eigenvectors to form D. We use a variant of McWeeny purification, called canonical purification [89], which allows us to compute D based on a given number of lowest energy eigenvalues (rather than the eigenvalue or energy level, in standard McWeeny purification). The basic algorithm is given below.

Algorithm 6: Canonical Purification Algorithm

```
1 Set D0
2 for k ← 1 to maxiter until convergence do
3 |   c_k = trace(D_k^2 − D_k^3)/trace(D^k − D_k^2)
4 |   if c_k ≤ 1/2 then
5 |   |   D_{k+1} = ((1 − 2c_k)D_k + (1 + c_k)D_k^2 − D_k^3)/(1 − c_k)
6 |   else
7 |   |   D_{k+1} = (1 + c_k)D_k^2 − D_k^3)/c_k
8 |   end
9 end
```

The purification algorithm spends most of its execution time in two matrix multiplications kernels: $S = D^2$ and $C = D^3$, where D is a symmetric real $N \times N$ matrix. We have

therefore focused on efficient distributed implementation of these two kernels. Note that purification performs many iterations and thus amortizes the cost of the initial data decomposition of D, C and S matrices across processors (nodes).

The 3D matrix multiply algorithm reduces communication bandwidth of the standard (2D) SUMMA algorithm [113] by arranging processors in 3D grid $p^{1/3} \times p^{1/3} \times p^{1/3}$, which can be viewed as $p^{1/3}$ number of $p^{1/3} \times p^{1/3}$ processor groups. While the details of the algorithm are well described in [28, 2], here we show our adaptation of this algorithm for our use-case of simultaneously computing S and C. Initially, processor group 0 contains a 2D distribution of D matrix. The algorithm proceeds in four steps:

1. Replicate D over $p^{1/3}$ processors groups
2. Compute $S_i = D \cdot D_i$ for i in $[1, p^{1/3}]$, where D_i is the i-th subset of columns, within processor group i. This involves broadcast of D_i along processor rows in the group, local dgemm, and reduction of $p^{1/3}$ computed columns into S_i
3. Compute $C_i = D \cdot S_i$ similarly, broadcasting S_i and performing local computation and reduction into C_i
4. Reduce corresponding S_i and C_i on processor group i onto group 0 for use by the remainder of the purification algorithm

The only communication performed by this algorithm is the reduction of S and C columns, and broadcast to spread the D input matrix. Hence the number of messages is $O(\log(p))$, similar to the SUMMA algorithm, but the bandwidth is $O(N^2/p^{2/3})$ which has lower asymptotic complexity by a factor of $p^{1/6}$ compared to the SUMMA algorithm.

3.5 Experimental Results

3.5.1 Experimental Setup

Performance results were measured on the Stampede supercomputer located at Texas Advanced Computing Center. The machine is currently (November 2014) ranked 7 on the

TOP500 list. We were able to use 1,024 nodes of the machine, which is the limit for jobs on the large queue. We used nodes composed of two Intel Sandy Bridge E5-2680 processors (8 cores each at 2.7 GHz) with one Intel Xeon Phi coprocessor (61 core). Memory on these nodes is 32 GB DRAM and 8 GB for the Intel Xeon Phi card.

Performance results were also measured on the Tianhe-2 supercomputer located at the National Supercomputing Center in Guangzhou, China. The machine was developed by the National University of Defense Technology, China, and is currently (November 2014) ranked first on the TOP500 list. Capable of a peak performance of 54.9 PFlops, Tianhe-2 has achieved a sustained performance of 33.9 PFlops with a performance-per-watt of 1.9 GFlops/W. Tianhe-2 has 1.4 PB memory, 12.4 PB storage capacity, and power consumption of 17.8 MW.

Tianhe-2 is composed of 16,000 nodes with a custom interconnect called TH Express-2 using a fat-tree topology. Each node is composed of two Intel Ivy Bridge E5-2692 processors (12 cores each at 2.2 GHz) and three Intel Xeon Phi 31S1P coprocessors (57 cores at 1.1 GHz). Memory on each node is 64 GB DRAM and 8 GB on each Intel Xeon Phi card.

All test molecules used the fully-contracted Dunning cc-pVDZ basis set [34]. A screening tolerance of $\tau = 10^{-10}$ was used. The system 1HSG is a human immunodeficiency virus (HIV) II protease complexed with a ligand (indinavir) which is a HIV inhibitor. We generated a set of test systems (Table 10) by only including residues a certain distance from any atom in the ligand. For a system named 1hsg_180, the distance is 18 Å. We also used test systems of approximately 1000 atoms, including an alkane, graphene, and DNA (a 19mer system).

3.5.2 Scaling Results

Figure 3 shows the scaling of execution time for four molecules. The experiments were performed on Stampede (CPU only). For each molecule, we compared the timing of parallel Fock matrix construction ("Fock"), purification ("Purif"), and eigendecomposition with

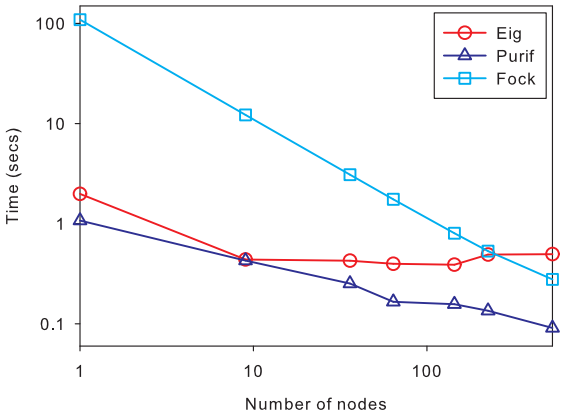
Table 10: Test molecules.

Molecule	Atoms	Shells	Basis Functions
1hsg_30	122	549	1159
1hsg_35	220	981	2063
1hsg_45	554	2427	5065
1hsg_70	789	3471	7257
1hsg_80	1035	4576	9584
1hsg_100	1424	6298	13194
1hsg_140	2145	9497	19903
1hsg_160	2633	11646	24394
1hsg_180	2938	13054	27394

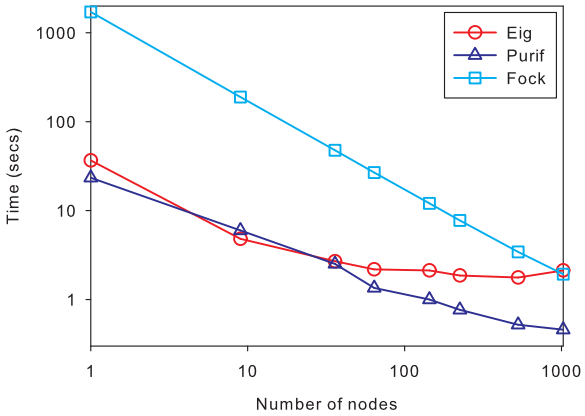
the pdsyevd function ("Eig"). The ScaLAPACK function pdsyevd implements the divide and conquer method for eigendecomposition, which is believed to scale better than the QR method.

As seen in the figure, Fock matrix construction shows good scaling for all molecules and dominates the HF performance on small numbers of nodes. For large numbers of nodes, the execution time of eigendecomposition can be larger than that of Fock matrix construction due to its worse scalability. For example, the eigendecomposition curve crosses the Fock matrix construction curve at 225 nodes for 1,159 basis functions, and at 1,024 nodes at 3,555 basis functions. It is interesting to note that for larger molecules the intersection of the Fock matrix construction curve and and the eigendecomposition curve appears to at a larger number of nodes. This is mainly because eigendecomposition scales better for larger molecules.

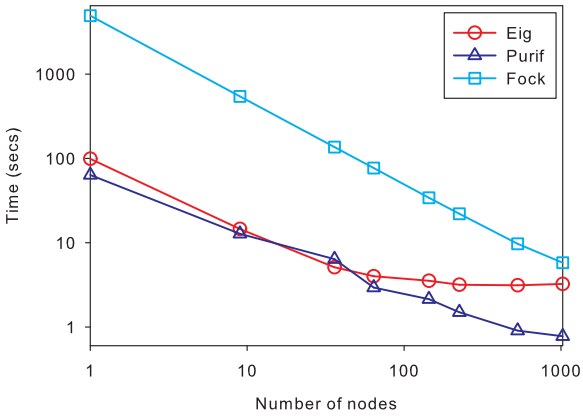
Figure 4 show the speedup of CPU-only calculations for four molecules on Stampede (CPU only). For each molecule, we measured the speedup of parallel Fock matrix construction ("Fock"), SCF using purification ("Fock + Purif"), and SCF using eigendecomposition ("Fock + Eig"). As see in the figure, SCF using purification scales better than SCF using eigendecomposition; both have better scalability for larger molecules. The loss in parallel efficiency due to purification is about the same size as the loss in efficiency due to Fock matrix construction alone. Thus one cannot say that scalability is impacted more



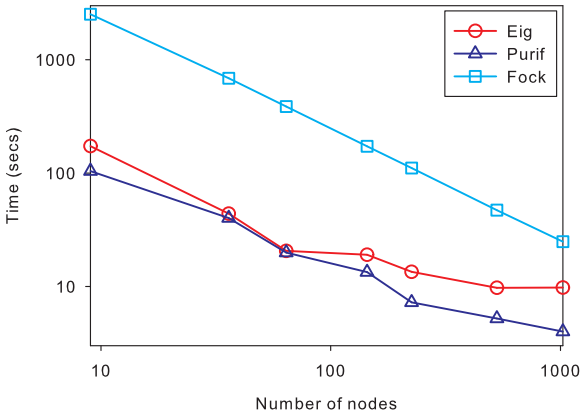
(a) 1hsg_30



(b) 1hsg_38



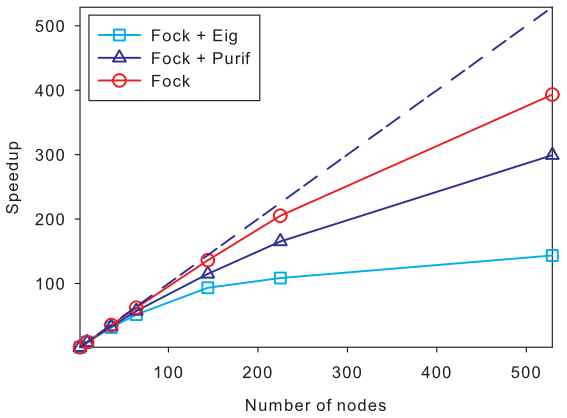
(c) 1hsg_45



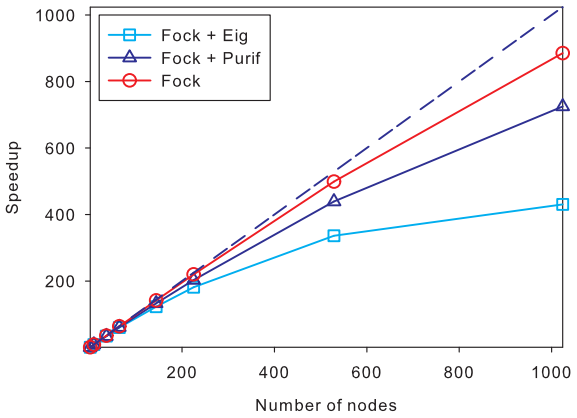
(d) 1hsg_90

Figure 3: Timing for Fock matrix construction and density matrix computation on Stampede (CPU only).

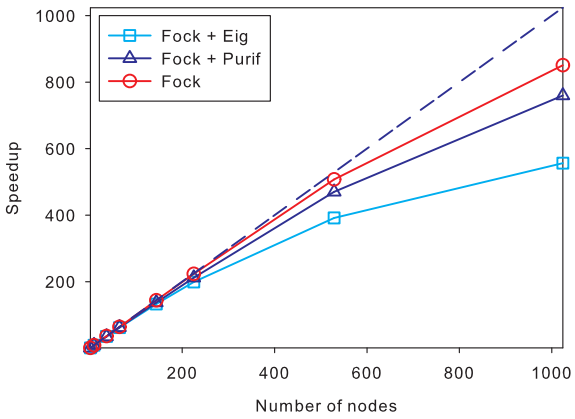
by Fock matrix construction or by purification; both impact the overall scalability by the same amount, due to the smaller amount of time spent in the less scalable density matrix calculation.



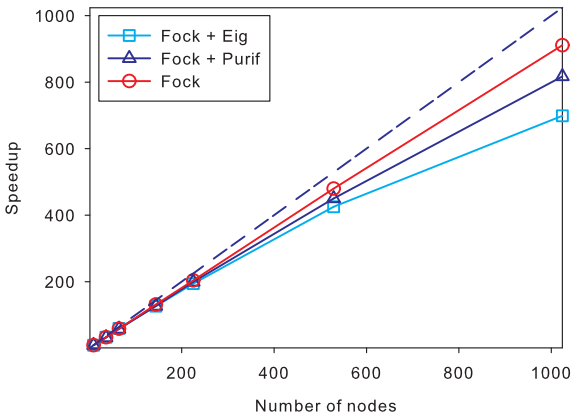
(a) 1hsg_30



(b) 1hsg_38



(c) 1hsg_45



(d) 1hsg_90

Figure 4: Scalability for Fock matrix construction and density matrix computation on Stampede (CPU only).

3.5.3 Comparison to NWChem

Figure 5 compares the performance of our HF code (GTFock) to NWChem for 1hsg_38. The experiments were performed on Stampede (CPU only). For both NWChem and GTFock, we measured the timing of parallel Fock matrix construction ("Fock") and the computation of density matrix ("Purif" for GTFock, "Eig" for NWChem). As seen in the figure, GTFock scales better than NWChem for both Fock matrix construction and density matrix calculation. For NWChem, Fock matrix construction scales up to about 144 nodes, but execution time increases with more nodes. Eigendecomposition, which requires only a small fraction of the execution time, scales poorly, and its execution time increases after 36 nodes. The maximum speedup of NWChem is 68x at 144 nodes. In comparison, GTFock achieves 142x speedup at 144 nodes.

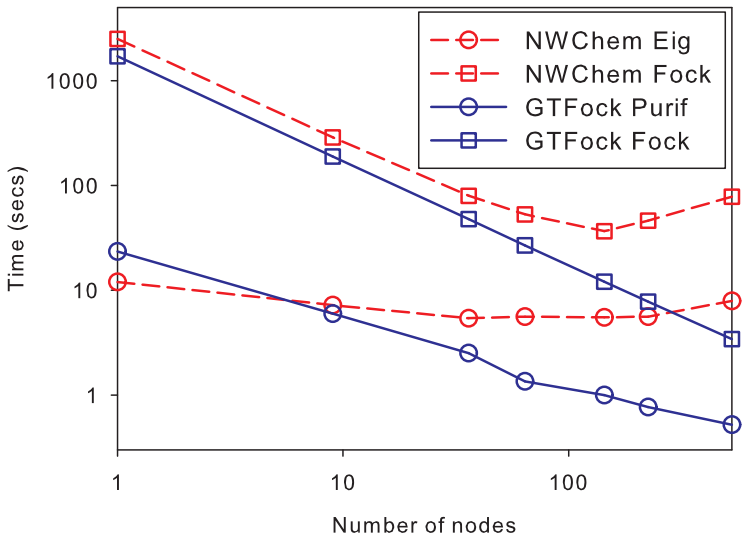


Figure 5: Comparison of NWChem to GTFock for 1hsg 38 on Stampede (CPU only).

3.5.4 HF Strong Scaling Results

Table 11 shows strong scaling results for the 1hsg_180 problem. We show results for both CPU-only calculations and for heterogeneous (CPU+coprocessor) calculations. Timings

are for a single SCF iteration. In the table, "Purif" denotes canonical purification for computing the density matrix, and "Fock" denotes Fock matrix construction, which involves integral calculations. Both steps involve distributed memory communication. "Comp" denotes the average computation time for integrals, and "GA" denotes the part of the Fock time spent in Global Arrays. We chose the number of nodes to be squares for convenience, but this is not necessary for our code. The 3D matrix multiplies used the nearest cube number of nodes smaller than the number of nodes shown.

In particular, Fock matrix construction shows very good speedup. An important observation is that purification timings are small relative to the timings for Fock matrix construction for this large problem. Also important is the observation that the purification time continues to decrease for increasing numbers of nodes up to 6,400. This is despite the fact that, as we increase the number of nodes, the dgemms performed by each node in the distributed matrix multiply algorithm become smaller and less efficient, while communication cost increases. Due to the increased inefficiency, the scaling of purification is much poorer than the scaling of Fock matrix construction. However, since timings for purification remain relatively small, they make a relatively small impact on total speedup.

Speedup at 8,100 nodes (CPU only; relative to 256 nodes) is 5954.1, or 73.5% parallel efficiency.When heterogenous mode is used, there is an additional 1.5× solution time improvement at 8100 nodes.

3.5.5 HF Weak Scaling Results

Weak scaling is difficult to measure because it is difficult to increase computational work exactly proportionally with number of nodes. This is primarily because, due to integral screening, the computational work is not known beforehand. However, we have chosen a set of molecules (Table 10) and chosen a number of nodes approximately equal to the square of the number of basis functions for each molecule. The performance results are shown in Table 12. To compute weak scaling, we scale the timings by the number of ERIs

Table 11: Timings (seconds) for 1hsg_180 on Tianhe-2. Top half of table is CPU-only mode; bottom half is heterogeneous mode.

Nodes	Fock	Comp	GA	Purif	Total
256	1772.6	1661.0	5.4	71.6	1845.4
576	820.4	738.1	6.7	34.9	859.9
1024	475.0	415.2	10.8	22.0	498.1
2500	197.9	170.0	8.1	15.6	214.4
4096	125.4	103.9	4.3	12.3	138.1
8100	70.8	52.7	5.6	8.2	79.3
4096	72.5	51.0	4.8	9.6	82.4
8100	44.1	26.2	3.1	8.4	52.9

computed. The results are shown in Figure 6. In the Figure, a plot for CPU-only mode is also shown (timing data not shown). Scalability is better for CPU-only because single node performance is worse.

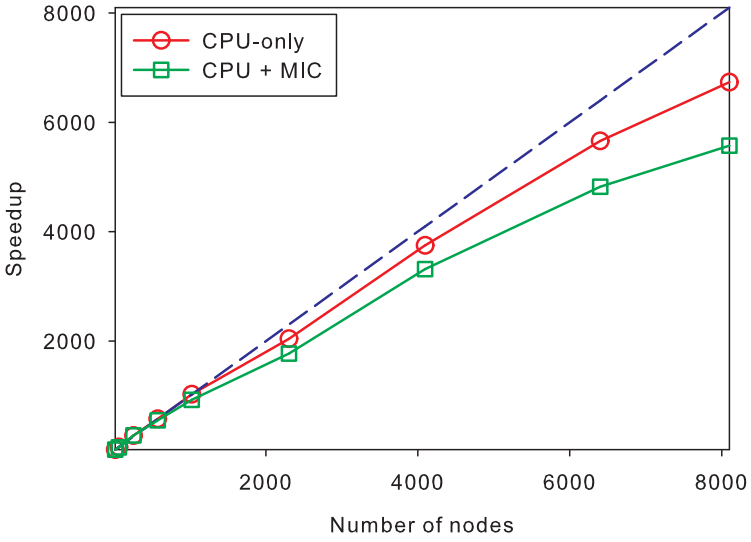


Figure 6: Weak scaling on Tianhe-2, relative to 16 nodes.

3.5.6 Flop Rate

Although SCF is not a flop-rich algorithm, it is still interesting to compute the achieved flop rate. We count the flops in purification and ERI calculation; all other operations (e.g.,

Table 12: Timing data (seconds) used to approximate weak scaling Tianhe-2. Top half of table is CPU-only mode; bottom half is heterogeneous mode.

Molecule	Nodes	Fock	Comp	Purif	Total
1hsg_30	16	14.02	13.39	0.33	14.39
1hsg_35	64	13.45	12.09	0.70	14.21
1hsg_45	256	40.09	37.15	1.34	41.51
1hsg_70	576	45.52	39.18	1.67	47.24
1hsg_80	1024	50.23	39.67	3.22	52.70
1hsg_100	2304	53.70	43.14	2.03	55.40
1hsg_140	4096	64.46	51.30	5.12	69.84
1hsg_160	6400	67.46	51.90	5.68	73.35
1hsg_180	8100	70.76	52.71	8.23	73.35
1hsg_30	16	7.2	6.6	0.4	7.9
1hsg_35	64	7.3	5.9	0.7	8.4
1hsg_45	256	21.1	18.2	1.2	22.7
1hsg_70	576	25.5	19.2	1.5	27.3
1hsg_80	1024	30.0	19.4	1.9	32.3
1hsg_100	2304	31.9	21.3	3.0	35.2
1hsg_140	4096	38.3	25.1	5.0	43.6
1hsg_160	6400	41.4	25.5	5.7	47.5
1hsg_180	8100	44.1	26.2	8.4	52.9

summation of the Fock matrix) perform a small number of flops, which we neglect. The number of flops spent in purification can be counted analytically. ERI calculation, however, is very unstructured, and the different types of integrals and different ways for integrals to be screened out makes analytical counting unwieldly. Instead, we use hardware counters to measure the number of flops in ERI calculations. Specifically, we use the *perf_events* interface exported by recent versions of the Linux kernel. As Xeon Phi does not have proper flop counters support, we perform all hardware event measurements on x86 CPUs. We compiled with the *-no-vec* option to avoid inaccuracies due to partial use of vector registers in SIMD operations. Results are shown in Table 13. We measured flops the following ways:

1. Retired floating point operations on AMD Piledriver, which separately counts multiplications and additions, and jointly counts divisions and square roots. We call this

count *intrinsic*. We verified the total of these counts using the retired flops counter on Intel Harpertown.

2. Floating point operations at the execution stage of the pipeline on Intel Nehalem. The counters we use also count compares, and may also overcount due to speculative execution and recirculations of these operations in the pipeline. We call this count *executed*, and it is an upper bound on the number of flops performed. This result gives additional confidence in our intrinsic count.

3. Xeon Phi does not have counters for flops. Also, Xeon Phi does not have single-instruction division and square root operations: these functions are computed with a sequence of multiplication, addition, and fused multiply-add operations. Square roots and divisions require a sequence of 10 and 11 flops, respectively, and thus the flop counts on Xeon Phi are higher than on CPUs. We instrumented our code to count the number of square root operations, and used AMD Piledriver counts to deduce the number of divisions. We used these results to estimate the number of flops performed on Xeon Phi.

Table 13: Flop counts (Gflops) for ERI calculation.

Molecule	Intrinsic	Executed	Xeon Phi
1hsg_30	8,550	9,612	12,443
1hsg_35	30,646	33,951	45,105
1hsg_45	386,695	448,561	575,323
1hsg_80	1,830,321	2,124,477	2,721,020
1hsg_140	8,751,659	10,223,033	13,027,801
1hsg_160	13,844,868	16,141,342	20,547,502
1hsg_180	17,820,050	20,853,142	26,487,829

Table 14 shows the flop rates using the timings from the previous tables. Interestingly, purification, which is based on dgemm, has a lower rate than ERI calculation. This is because of the small size of the matrices per node, as well as communication costs. In

summary, for the largest problem on 8100 nodes, the aggregate flop rate for SCF is 441.9 Tflops/s.

Table 14: Flop rates (Tflops/s) for Table 12.

	Nodes	ERI	Purif	Total SCF
1hsg_30	16	1.7	0.5	1.4
1hsg_35	64	6.6	1.7	4.9
1hsg_45	256	27.5	14.2	22.6
1hsg_70	576	62.3	34.6	44.9
1hsg_80	1024	121.7	63.3	75.2
1hsg_100	2304	230.0	107.3	142.2
1hsg_140	4096	450.0	222.4	264.7
1hsg_160	6400	701.3	356.2	383.6
1hsg_180	8100	879.2	352.6	441.9

3.6 Summary

We have presented an efficient implementation of HF calculations on large-scale distributed systems. To the best of our knowledge, we have used the largest machine configuration (half of Tianhe-2) ever used for HF calculations, and our HF code is able to compute 1.64 trillion electron repulsion integrals per second. For 1hsg_45 on only 576 nodes, our time to solution is 10.1$\times$ faster than for NWChem, and the improvement is expected to be greater for more nodes.

There are several performance limitations of ERI calculation. Most clearly, there is a need to improve SIMD operation. If one node computes all integrals, sorting the integrals by type can lead to abundant SIMD parallelism. However, for distributed computations, it may be worthwhile to distribute batches of shell quartets with the same integral type. This would allow better SIMD operation on each node. The tradeoff is potentially worse communication patterns when shell quartets are distributed this way. In addition to small loop counts hurting SIMD performance, other likely performance limitations of ERI calculation include unpredictable control flow (screening), high register pressure, and indirect memory access (data referenced through indices).

Because of the similarity between Hartree-Fock theory and Kohn-Sham density functional theory (DFT), the work presented here can be readily extended to DFT. Moreover, additional quantum mechanical methods may be formulated in terms of (generalized) Coulomb and Exchange matrices, meaning that this work may form the core of future massively parallel codes for methods including symmetry-adapted perturbation theory (SAPT), configuration interaction singles (CIS) for excited electronic states, coupled-perturbed Hartree-Fock or DFT for analytical energy gradients, and others.

Chapter IV

"MATRIX-FREE" ALGORITHM FOR HYDRODYNAMIC BROWNIAN SIMULATIONS

From this chapter, we will proceed to present the high performance algorithms and software for large-scale Brownian and Stokesian simulations. In this chapter, we will describe a matrix approach for hydrodynamic Brownian simulations, which has both lower computational complexity and memory requirement than the conventional approaches.

Brownian dynamics (BD) is a computational method for simulating the motion of particles, such as macromolecules or nanoparticles, in a fluid environment. It has been widely used in multiple areas including biology, biochemistry, chemical engineering and material science. Currently, many BD simulations ignore the effect of the long-range hydrodynamic interactions (HI) between particles in a fluid. Although this choice is made to reduce the computational cost of these simulations, the modeling of HI makes BD simulations more realistic and more comparable to experiments. Efficient ways of modeling HI are arguably one of the biggest hurdles facing computational biologists striving for higher fidelity macromolecular simulations [45].

Conventional BD simulations with HI utilize $3n \times 3n$ dense mobility matrices, where n is the number of simulated particles. This limits the size of BD simulations, particularly on accelerators with low memory capacities. To address this problem, we will present a new matrix-free algorithm for BD simulations, allowing us to scale to very large numbers of particles while also being efficient for small numbers of particles. The matrix-free approach also allows large-scale BD simulations to be accelerated on hardware that have relatively low memory capacities, such as GPUs and Intel Xeon Phi. We will also describe an efficient implementation of this method for multicore and manycore architectures, as

well as a hybrid implementation that splits the workload between CPUs and Intel Xeon Phi coprocessors.

4.1 Background: Conventional Ewald BD Algorithm

4.1.1 Brownian Dynamics with Hydrodynamic Interactions

In BD simulations, the solvent molecules are modeled as spherical particles of possibly varying radii. Like in other particle simulation methods, particle positions are propagated step by step. The BD propagation formula with HI can be expressed as [38]

$$\vec{r}(t+\Delta t) = \vec{r}(t) + M\vec{f}\Delta t + k_B T(\nabla \cdot M)\Delta t + \vec{g} \tag{13}$$

$$\langle \vec{g} \rangle = 0, \quad \langle \vec{g}\,\vec{g} \rangle = 2k_B T M \Delta t.$$

Here, $\vec{r}$ is the position vector of the n particles, t is the time, Δt is the time step length, k_B is Boltzmann's constant, T is the temperature, $\vec{f}$ is the forces determined by the gradient of potential energy, and $\vec{g}$ is the Brownian displacement. The term involving $\vec{f}$ may represent van der Waals or bonded interactions, etc.

The matrix M is the *mobility matrix*. The (i, j)-th entry in M is a 3×3 tensor describing the interaction between particles i and j. Thus, the size of M is $3n \times 3n$. The Rotne-Prager-Yamakawa [99, 122] (RPY) tensor is widely used for modeling HI in BD simulations. With free boundary conditions, the entries of M using the RPY tensor are

$$M_{ij} = \frac{1}{6\pi\eta a}\left[\frac{3a}{4\|\vec{r}_{ij}\|}(I + \hat{r}_{ij}\hat{r}_{ij}^T) + \frac{a^3}{2\|\vec{r}_{ij}\|^3}(I - 3\hat{r}_{ij}\hat{r}_{ij}^T)\right] \tag{14}$$

when $i \neq j$, and $M_{ii} = (6\pi\eta a)^{-1} I$ otherwise. In the above, $\vec{r}_{ij}$ is the vector between particles i and j, $\hat{r}_{ij}$ is the normalized vector, η is the viscosity, and a is the radii of the particles. The matrix M is symmetric positive definite for all particle configurations. As is common, we use conditions such that $\nabla \cdot M = 0$ so that the third term in Equation (13) is zero.

4.1.2 Ewald Summation of the RPY Tensor

Periodic boundary conditions are widely used in BD simulations. In this case, a particle i not only has long-range interactions with a particle j in its own simulation box, but also

with all the images of j in the infinite number of replicas of the simulation box tiling all of space. In matrix terms, this means that the tensor M_{ij} describing the interaction between particles i and j is an infinite sum of terms.

Ewald summation is the standard procedure for computing infinite sums, by reformulating the sum into two rapidly converging sums, one in real space and one in reciprocal (Fourier) space. The Ewald sum has the form

$$M = M^{real} + M^{recip} + M^{self}$$

where the first term is the real-space sum, the second-term is the reciprocal-space sum, and the third term is a constant. For the RPY tensor, Beenakker [14] derived the formulas

$$\begin{aligned} M_{ij}^{real} &= \sum_{\vec{l}} M_{\alpha}^{(1)}(\vec{r}_{ij} + \vec{l}L) \\ M_{ij}^{recip} &= \frac{1}{L^3} \sum_{\vec{k} \neq 0} \exp(-i\vec{k} \cdot \vec{r}_{ij}) M_{\alpha}^{(2)}(\vec{k}) \\ M_{ij}^{self} &= M_{\alpha}^{(0)} \delta_{ij} \end{aligned}$$

where L is the width of the simulation box and δ_{ij} is the Kronecker delta. The real-space sum is over all replicas $\vec{l}$ of the simulation box ($\vec{l}$ is a vector from the simulation box to one of its replicas). The reciprocal-space sum is over all Fourier lattice vectors $\vec{k} \neq 0$. The functions $M_{\alpha}^{(1)}(\vec{r})$ and $M_{\alpha}^{(2)}(\vec{k})$ derived by Beenakker [14] are designed to decay quickly with $\|\vec{r}\|$ and $\|\vec{k}\|$, respectively.

These functions are parameterized by α (*Ewald parameter*); if α is large, then the real-space sum converges faster than the reciprocal-space sum, and vice-versa. Thus α tunes the amount of work between the real-space and reciprocal-space sums and can be chosen to reduce computation time depending on the relative cost of performing the two summations. In practice, hundreds of vectors $\vec{l}$ and $\vec{k}$ are needed for reasonable accuracy of the sums, making the complexity of Ewald summation $O(n^2)$ with a large constant. Periodic boundary conditions thus make the mobility matrix M very costly to compute.

4.1.3 Brownian Displacements

From Equation (13), the Brownian displacement $\vec{g}$ is a random vector from a multivariate Gaussian distribution with mean zero and covariance $2k_BTM\Delta t$. This covariance is required by the fluctuation-dissipation theorem, relating stochastic forces with friction. The standard way to calculate $\vec{g}$ is by

$$\vec{g} = \sqrt{2k_BT\Delta t}\, S\, \vec{z}$$

where S is the lower-triangular Cholesky factor of M,

$$M = SS^T$$

and $\vec{z}$ is a standard Gaussian random vector. The correlated vector $\vec{y} = S\vec{z}$ has the Gaussian distribution $N(0, M)$. Any S that satisfies $M = SS^T$ may be used and it is common to choose S as the lower-triangular Cholesky factor of M.

4.1.4 Ewald BD Algorithm

The BD algorithm with HI using Ewald summation and Cholesky factorization is presented in Algorithm 7. An observation used in BD simulations is that the mobility matrix changes slowly over time steps, meaning it is possible to use the same matrix for many time steps. Let λ_{RPY} be the update interval for the mobility matrix, whose value usually ranges from 10 to 100. In Algorithm 7, the mobility matrix and the Cholesky factorization are constructed only every λ_{RPY} time steps, and λ_{RPY} Brownian displacement vectors are generated together. Algorithm 7, called the *Ewald BD algorithm* is the baseline algorithm we use for comparisons.

4.2 Motivation

Computing BD with HI is very computationally expensive, and this is mainly due to two separate calculations at each time step of BD simulations (see Algorithm 7): 1) the construction of a dense $3n \times 3n$ hydrodynamic mobility matrix, M, which is a function of the

Algorithm 7: Standard BD algorithm for m time steps. $\vec{r}_k$ denotes the position vector at time step k.

$k \leftarrow 0$
$n \leftarrow m/\lambda$
for $i \leftarrow 1$ **to** n **do**
    Construct dense matrix $M_0 = M(\vec{r}_k)$ using Ewald sums
    Compute Cholesky factorization $M_0 = SS^T$
    Generate λ random vectors $Z = [\vec{z}_1, \cdots, \vec{z}_\lambda]$
    Compute $D = [\vec{d}_1, \cdots, \vec{d}_\lambda] = \sqrt{2k_BT\Delta t}\, SZ$
    for $j \leftarrow 1$ **to** λ **do**
        Compute $\vec{f}(\vec{r}_k)$
        Update $\vec{r}_{k+1} = \vec{r}_k + M_0\, \vec{f}(\vec{r}_k)\Delta t + \vec{d}_j$
        $k \leftarrow k+1$
    end
end

configuration of the n particles, and 2) the calculation of a Brownian displacement vector which must come from a Gaussian distribution with covariance proportional to M. The construction of M is especially costly in the usual case when periodic boundary conditions are used, due to the need to approximate infinite sums using Ewald summation, an $O(n^2)$ algorithm with a very large constant. The calculation of a Brownian displacement vector requires computing some form of square root of M. The simplest and most common technique is to compute a Cholesky factorization, requiring the matrix M to be available explicitly, and which has computational complexity $O(n^3)$. The combination of $O(n^2)$ storage and $O(n^3)$ computation limits the size of BD simulations with HI.

In order to address the challenges of large-scale BD simulations with HI, we propose a new matrix-free approach. The main idea is to avoid constructing the hydrodynamic mobility matrix, and to utilize methods for computing Brownian displacements that do not require this matrix to be available explicitly. Specifically, we replace the mobility matrix by a particle-mesh Ewald (PME) summation. The PME algorithm is well-known for computing electrostatic interactions in molecular dynamics (MD) simulations [26, 39]. It scales as $O(n \log n)$ and only requires $O(n)$ storage. To compute Brownian displacements

without a matrix in this context, we use a Krylov subspace method [6].

This work is partially motivated by the desire to utilize the accelerating hardware. Accelerators, such as GPU cards and Intel Xeon Phi coprocessors, are limited by relatively low memory capabilities. However, the Ewald BD algorithm requires $O(n^2)$ storage, which limits its use on accelerators. For instance, an existing GPU code for BD, called BD_BOX [33], is limited to approximately 3,000 particles due to memory restrictions.

4.3 Related Work

Existing BD codes that model HI, including BD_BOX [33] and Brownmove [47], use the conventional Ewald BD algorithm, in which the mobility matrix is explicitly constructed. There also exist BD codes optimized for GPUs [33, 95], but they are limited to approximately 3,000 particles, again due to the explicit construction of the dense mobility matrix. HI can also be modeled using a sparse matrix approximation [110, 11] when the interactions are primarily short-range; this is the approach used in LAMMPS [93].

Matrix-free approaches for particle-fluid simulations is not completely new. PME has been used to accelerate simulations in Stokesian conditions [52, 104, 101]. However, these codes use the PME summation of the Stokeslet or Oseen tensor, rather than the Rotne-Prager-Yamakawa [99, 122] tensor widely used in BD. In our work, we use this latter tensor with PME and also specifically handle the computation of Brownian forces with PME.

PME algorithms are implemented widely in molecular dynamics codes for electrostatic interactions. Harvey and Fabritiis [54], describe an implementation of smooth PME on GPU hardware for MD. However, there are many differences between PME for MD and PME for BD that affect implementation and use.

4.4 Matrix-Free BD Algorithm

In this section, the novel matrix-free BD algorithm is present. First, we will derive the PME equations for the RPY tensor, which is one of the main contributions of our work.

4.4.1 Particle-Mesh Ewald for the RPY Tensor

The PME method is a fast method for computing Ewald summations. The main idea is to use FFTs, rather than standard discrete transforms, to sum the reciprocal-space part of the Ewald summation. Since the particles are not regularly spaced, particle forces are first interpolated onto a regular mesh. The computed velocities on the regular mesh are then interpolated back onto the original particle locations.

The RPY operator acting on a vector of forces $\vec{f}$ can be written as

$$M\vec{f} = \underbrace{M^{real}\vec{f}}_{\vec{u}^{real}} + \underbrace{M^{recip}\vec{f}}_{\vec{u}^{recip}} + \underbrace{M^{self}\vec{f}}_{\vec{u}^{self}}$$

where the right-hand side consists of the real-space term, the reciprocal-space term, and a self term. The M^{self} operator is a constant tensor scaling, which is cheap and easy to implement in parallel; it will not be discussed further in this paper. In PME, the Ewald parameter α is chosen so that the real-space term can be computed using interactions between particles within a small cutoff distance r_{max}. The M^{real} operator can then be regarded as a sparse matrix with nonzeros M_{ij}^{real} when particles i and j are separated by not more than r_{max}. Since we will need to apply M^{real} multiple times to different vectors, it is advantageous to store it as a sparse matrix and perform the computation of the real-space term u^{real} as a sparse matrix-vector product (SpMV). We still call our approach "matrix-free" although we do construct this one sparse matrix.

Recall that the reciprocal-space part of an Ewald summation is given by

$$M_{ij}^{recip} = \frac{1}{L^3}\sum_{\vec{k}\neq 0}\exp(-i\vec{k}\cdot\vec{r}_{ij})M_{\alpha}^{(2)}(\vec{k}) \tag{15}$$

Utilizing Equation (15), the reciprocal-space term $M^{recip}\vec{f}$ is

$$\vec{u}_j^{recip} = \frac{1}{L^3}\sum_{\vec{k}\neq 0}\sum_{i=1}^{n}\exp(-i\vec{k}\cdot\vec{r}_{ij})M_{\alpha}^{(2)}(\vec{k})\vec{f}_i$$

where $\vec{f}_i$ is the force on the i-th particle. By a simple manipulation,

$$\vec{u}_j^{recip} = \frac{1}{L^3}\sum_{\vec{k}\neq 0}\exp(i\vec{k}\cdot\vec{r}_j)M_{\alpha}^{(2)}(\vec{k})\sum_{i=1}^{n}\exp(-i\vec{k}\cdot\vec{r}_i)\vec{f}_i$$

where $\vec{r}_i$ and $\vec{r}_j$ are the positions of the i-th and j-th particles, respectively. Let us denote

$$\vec{f}^{recip}(\vec{k}) = \sum_{i=1}^{n} \exp(-i\vec{k}\cdot\vec{r}_i)\vec{f}_i$$

which can be regarded as the Fourier transform of the forces $\vec{f}$. We are now able to reuse $\vec{f}^{recip}(\vec{k})$ for computing $\vec{u}_j^{recip}$ for all particles j.

The main benefit of PME over Ewald is to sum complex exponentials using the FFT. This cannot be achieved as written above because the $\vec{r}_i$ are not equally spaced. Therefore, the PME method first spreads the forces (the spreading operation is the transpose of interpolation) onto a regular mesh in order to compute $\vec{f}^{recip}(\vec{k})$ via the FFT. Then, after multiplying $\vec{f}^{recip}(\vec{k})$ by $M_\alpha^{(2)}(\vec{k})$, the velocity is

$$\vec{u}_j^{recip} = \frac{1}{L^3} \sum_{\vec{k}\neq 0} \exp(i\vec{k}\cdot\vec{r}_j) M_\alpha^{(2)}(\vec{k}) \vec{f}^{recip}(\vec{k}). \tag{16}$$

The sum over lattice vectors $\vec{k}$ corresponds to an inverse Fourier transform. Again, the result is computed on the regular mesh. The velocities computed on the mesh are finally interpolated onto the locations of the particles. In the case where particles have different radii, the above procedure is still applicable, given a pre-scaling of the point forces and a post-scaling of the particle velocities.

We now introduce the interpolation method. Different interpolation methods give rise to different PME variants, the major ones being PPPM [55], PME [26] and SPME. We use cardinal B-spline interpolation, as used in smooth PME (SPME) [39]. We found the SPME approach to be more accurate than the original PME approach [26] with Lagrangian interpolation, while negligibly increasing computational cost. In contrast to SPME with a scalar kernel such as for electrostatics, SPME with a 3×3 tensor kernel means that spreading is applied to 3×1 quantities. Let the force at particle i be denoted by $\vec{f}_i = [f_i^x, f_i^y, f_i^z]$. These forces are spread onto a $K\times K\times K$ mesh,

$$F^\theta(k_1,k_2,k_3) = \sum_{i=1}^{N} \vec{f}_i^\theta \; W_p(u_i^x - k_1) W_p(u_i^y - k_2) W_p(u_i^z - k_3) \tag{17}$$

where θ can be x, y or z, and u^x, u^y and u^z are scaled fractional coordinates of the particles, i. e., for a given particle located at (r^x, r^y, r^z), we have $u^\theta = r^\theta K/L$ for a cubical $L \times L \times L$ simulation box. The functions W_p are cardinal B-splines of order p (piecewise polynomials of degree $p-1$). For simplicity of notation, we assume that the W_p functions "wrap around" the periodic boundaries. Importantly, the W_p have compact support: they are nonzero over an interval of length p. Thus Equation (17) shows that each $\vec{f}_i^\theta$ is spread onto p^3 points of the mesh around the i-th particle. Figure 7 illustrates the spreading of a force onto a 2D mesh.

In practice, the 3D meshes and the spreading forces are stored in 1D arrays – this is required for the FFT libraries. However, for the sake of convenience, we use 3D indexing notation to index the arrays.

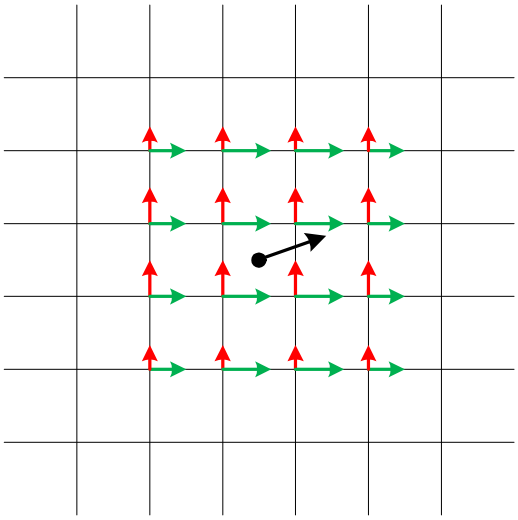


Figure 7: Spreading a force onto a 2D mesh using B-splines of order 4. The black dot represents a particle, and the black arrow represents a force on the particle. The spread forces on the mesh are represented by the red and green arrows.

After spreading the forces into the regular mesh, we can apply the FFT. The approximation to the θ component of $\vec{f}^{recip}(\vec{k})$ on the mesh is $b_1(k_1)b_2(k_2)b_3(k_3)\mathscr{F}[F^\theta(k_1,k_2,k_3)]$ where $\mathscr{F}$ denotes the 3D FFT, and where the b functions are complex scalars which can be interpreted as coefficients of interpolating complex exponentials with B-splines (unnecessary when PME Lagrangian interpolation is used). The b functions are fixed given the

interpolation scheme and have negligible computational cost. In practice, they are absorbed into the influence function to be described next, and we will not further discuss them in this paper.

Referring to Equation (16), the next step is to multiply the Fourier transform of the forces by $M_\alpha^{(2)}(\vec{k})$. Assuming uniform particle radii, $M_\alpha^{(2)}(\vec{k})$ derived in [14] is

$$\begin{aligned} M_\alpha^{(2)}(\vec{k}) =& (I - \hat{k}\hat{k}^T) m_\alpha(\|\vec{k}\|) \\ m_\alpha(\|\vec{k}\|) =& (a - \tfrac{1}{3}a^3\|\vec{k}\|^2)(1 + \tfrac{1}{4}\alpha^{-2}\|\vec{k}\|^2 + \tfrac{1}{8}\alpha^{-4}\|\vec{k}\|^4) \\ & \times 6\pi\|\vec{k}\|^{-2}\exp(-\tfrac{1}{4}\alpha^{-2}\|\vec{k}\|^2) \end{aligned} \tag{18}$$

where $\hat{k}$ is the normalization of $\vec{k}$, a is the radius of the particles, $(I - \hat{k}\hat{k}^T)$ is a 3×3 tensor, and $m_\alpha(\|\vec{k}\|)$ is a scalar function. Computationally, $M_\alpha^{(2)}(\vec{k})$ is a 3×3 symmetric matrix defined at each of the $K \times K \times K$ mesh points. We refer to this structure as the *influence function*, $I(k_1, k_2, k_3)$. Defining $C^\theta(k_1, k_2, k_3) = \mathscr{F}[F^\theta(k_1, k_2, k_3)]$, applying the influence function means computing

$$D^\theta(k_1, k_2, k_3) = I(k_1, k_2, k_3) \cdot C^\theta(k_1, k_2, k_3). \tag{19}$$

The velocities on the mesh can be computed as

$$U^\theta(k_1, k_2, k_3) = \mathscr{F}^{-1}[D^\theta(k_1, k_2, k_3)]$$

where $\mathscr{F}^{-1}$ denotes the 3D inverse FFT. The particle velocities $\vec{u}^{recip}$ can be computed by interpolating U^θ onto the locations of the particles, which is the reverse process of spreading.

In summary, each PME operation effectively multiplies the mobility matrix by a given vector of forces $\vec{f}$ for the particles located at $\vec{r}$. In the following, we will use $\vec{u} = PME(\vec{f})$ to denote this operation, given some particle configuration $\vec{r}$. The creation of a PME operator in software includes the construction of the sparse matrix M^{real} and other pre-processing steps needed for applying M^{recip}.

4.4.2 Computing Brownian Displacements with PME

The canonical method of computing Brownian displacements, which uses Cholesky factorization, requires M to be available as a matrix. With PME, such an explicit form for M is not available.

It is possible to compute a Brownian displacement vector without the need to have M in matrix form. An approximate correlated vector may be computed as

$$y = p_0 z + p_1 M z + p_2 M^2 z + \cdots + p_{m-1} M^{m-1} z = p(M) z$$

where $p(M)$ is a Chebyshev polynomial that approximates the principal square root of M [42]. The matrix $p(M)$ is not formed, and thus M is not required to be available as a matrix; instead, it is only necessary to apply M as an operator on a vector. This method allows us to use any fast method of applying this operator. In the current literature, M is simply applied as a dense matrix-vector product. A disadvantage of the Chebyshev polynomial method is that estimates of the spectrum of M are needed, which can be costly since M changes during a BD simulation [48].

A more efficient way is to use Krylov subspace methods [6, 5]. These methods only require the ability to perform matrix-vector products with M and do not require eigenvalue estimates of M. Thus this method is suitable for computing Brownian displacements in the PME context. In these methods, an approximation to the square root of a matrix times a vector, $M^{1/2}\vec{z}$, is constructed from the Krylov subspace

$$K_m(M, \vec{z}) = \text{span}\{\vec{z}, M\vec{z}, \ldots, M^{m-1}\vec{z}\}$$

where m is the dimension of the subspace. These methods can be extended to compute a block of Brownian displacement vectors simultaneously, using a block Krylov subspace

Since in BD we can use the same mobility matrix for several time steps, we use a *block* Krylov subspace method to compute Brownian displacement vectors for multiple time steps simultaneously. This has the benefit of (a) fewer total number of iterations are required for

convergence than the single vector Krylov method, leading to lower computational cost per vector [6], and (b) the SpMV operation for computing the real-space term is applied to a block of vectors, which is more efficient than single vector SpMV [71].

4.4.3 Matrix-Free BD Algorithm

The matrix-free BD algorithm is shown in Algorithm 8. Krylov(PME, Z) denotes an application of the Krylov subspace method, in which the products of the mobility matrix with a given vector $\vec{f}$ are evaluated by $PME(\vec{f})$. Since the mobility matrix changes slowly over time steps, we can use the particle configuration at the current time step to compute the Brownian displacement vectors for the following λ_{RPY} time steps. The sparse matrix M^{real} is only updated (at line 4) every λ_{RPY} time steps.

Algorithm 8: Matrix-free BD algorithm for m time steps. $\vec{r}_k$ denotes the position vector at time step k.

```
1  k ← 0
2  n ← m/λ_RPY
3  for i ← 1 to n do
4  |   Construct PME operator using r_k
5  |   Generate λ_RPY random vectors Z = [z_1, ··· , z_λRPY]
6  |   Compute D = [d_1, ··· , d_λRPY] = Krylov(PME, Z)
7  |   for j ← 1 to λ_PRY do
8  |   |   Compute f(r_k)
9  |   |   Update r_{k+1} = r_k + PME(f(r_k))Δt + d_j
10 |   |   k ← k + 1
11 |   end
12 end
```

4.5 Hybrid Implementation of PME

4.5.1 Reformulating the Reciprocal-Space Calculation

In molecular dynamics simulations, PME is not applied more than once for a given particle configuration. In our work, we apply PME iteratively in Krylov subspace methods to compute Brownian displacements. Thus our optimization of PME involves a setup phase where precomputation is used to speed up the actual PME computations.

In this section, we reformulate the interpolation and spreading operations in the reciprocal-space calculation of PME as sparse matrix-vector products. Also, for our tensor kernel, applying the influence function can be regarded as a matrix-vector product for a block diagonal matrix with 3×3 blocks. The breakdown of the reciprocal-space calculation into these kernels also allows us to implement an efficient PME code in a relatively portable way.

Define the $n \times K^3$ interpolation matrix P as

$$P(i,\ k_1K^2 + k_2K + k_3) = W_p(u_i^x - k_1)W_p(u_i^y - k_2)W_p(u_i^z - k_3) \tag{20}$$

which is a sparse matrix with p^3 nonzeros per row (we have assumed 1-based indexing for i and 0-based indexing for k_1, k_2, k_3). The spreading of forces can then be expressed as

$$[F^x, F^y, F^z] = P^T \times [\vec{f}^x, \vec{f}^y, \vec{f}^z] \tag{21}$$

and, similarly, the interpolation of velocities is

$$[(\vec{u}^{recip})^x, (\vec{u}^{recip})^y, (\vec{u}^{recip})^z] = P \times [U^x, U^y, U^z]. \tag{22}$$

After the reformulation, the reciprocal-space calculation of PME can be performed in six steps:

(1) *Constructing P*: Precomputation of the interpolation matrix P (Equation (20)).

(2) *Spreading*: Spreading of the forces onto the mesh array F^θ (Equation (21)). This has been reformulated as a sparse matrix-vector product.

(3) *Forward FFT*: Applying 3D fast Fourier transform to compute $C^\theta = \mathscr{F}[F^\theta]$.

(4) *Influence function*: Multiplying C^θ by the influence function I (Equation (19)).

(5) *Inverse FFT*: Applying 3D inverse fast Fourier transform $U^\theta = \mathscr{F}^{-1}[D^\theta]$.

(6) *Interpolation*: Interpolating the velocities on the locations of the particles (Equation (22)).

There are two main benefits of this reformulation. First, our matrix-free BD algorithm uses the Krylov subspace method to compute Brownian displacements, which requires computing PME multiple times at each simulation step. Since P only depends on

the positions of the particles, with this reformulation we only need to precompute P once at the beginning of each simulation step when the PME operator is constructed (line 4 in Algorithm 8), and reuse it for all the PME computations within the step. This significantly reduces the computational cost. In addition, this reformulation transforms spreading and interpolation into SpMV operations for which high-performance implementations are available, including on accelerators.

4.5.2 Optimizing the Reciprocal-Space Calculation

4.5.2.1 Constructing P

The precomputation of the interpolation matrix P is performed in parallel, with P partitioned into row blocks, one for each thread. SIMD instructions are used by each thread to compute multiple rows concurrently in a row block. It is natural to store P in Compressed Sparse Row (CSR) format. However, the row pointers are not necessary since all rows of P have the same number of nonzeros (each force is spread onto the same number of FFT mesh points).

4.5.2.2 Spreading

The spreading step multiplies the transpose of P by the vector of forces. Since P is stored in CSR format, different threads will try to update the same memory locations in the result. To alleviate this contention, one might explicitly transpose P and store it in Compressed Sparse Column (CSC) format. However, CSC format can be inefficient for storing P since P is typically "short-and-fat" and contains many empty columns (corresponding to mesh points that receive no spreading contributions from particles).

In order to efficiently parallelize the P^T operation, we partition the mesh points into square blocks with dimensions no less than $p \times p$, where p is the interpolation order. Those blocks are then partitioned into groups such that there are no two blocks in one group that are adjacent to each other. We call such a group an *independent set*. There are eight independent sets in a 3D mesh. By observing that the particles from different blocks in the

same independent set own distinct columns of P, the forces for those particles can be spread in parallel without write contention. Figure 8 illustrates independent sets in a 2D mesh, in which different independent sets are shown in different colors. In our parallel spreading implementation, the particles are first mapped into the blocks. The parallel SpMV is then performed in eight stages, and each stage only multiplies the rows of P that are associated with the particles from one independent set.

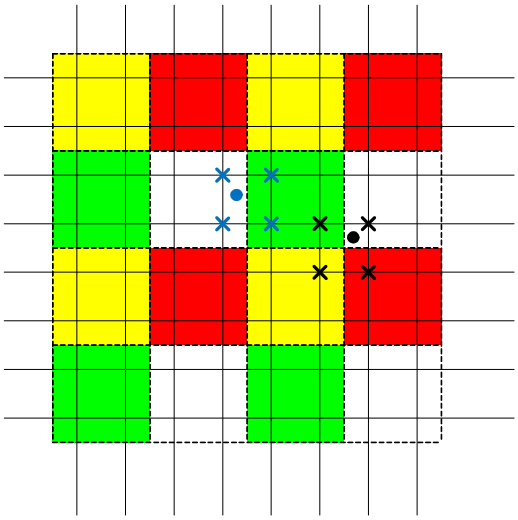


Figure 8: Independent sets for $p = 2$ in a 2D mesh. The 4 independent sets (8 in 3D) are marked in different colors. Two particles (dots) from different blocks in the same independent set cannot spread to the same mesh points (crosses).

Note that for a given set of particle positions at a time step, some mesh points may have no contribution from particles. Thus we explicitly set the result F^{θ} to zero before beginning the spreading operation.

4.5.2.3 3D FFTs

We use the Intel MKL to perform 3D FFTs. The library contains in-place real-to-complex forward FFT and complex-to-real inverse FFT routines. This halves the memory and bandwidth requirements compared to the case if only complex-to-complex routines were available. Similarly, only half of the influence function is needed, which can be stored in an array of size $K \times K \times (K/2+1)$, where each array entry itself is a symmetric 3×3 matrix.

4.5.2.4 *Applying the Influence Function*

The influence function I only depends on the simulation box size L, the mesh size K, and the interpolation order p; it can thus be precomputed and used for any particle configuration. However, the memory required for explicitly storing I is approximately $6 \times 8 \times K^3/2$ bytes, which is 3 times larger than the storage requirement for F^θ, making this approach impractical for Intel Xeon Phi and other accelerators that have limited memory. On the other hand, constructing I on-the-fly, every time it is needed, requires evaluating exponential functions, which are costly to compute.

We observe from Equation (18), however, that the influence function is the product of a 3×3 tensor and a relatively expensive scalar function, m_α. We precompute and store this scalar function rather than the 3×3 tensor, giving a savings of a factor of 6. When applying the influence function, the quantity $(I - \hat{k}\hat{k}^T)$, which only depends on the lattice vector, can be constructed without memory accesses. Applying the influence function is memory bandwidth bound, due to the small number of flops executed compared to data transferred for reading C^θ and writing D^θ. This procedure would be compute-bound if the influence function were computed on-the-fly.

Each application of a tensor of I on C^θ computes the product of a 3×3 matrix with 3 complex values in double precision, which requires 36 floating point operations and 104 bytes memory traffic.

4.5.2.5 *Interpolation*

In our reformulation, interpolating the velocities from the mesh points is performed by a sparse matrix-vector product. For this, we have implemented a parallel SpMV kernel for multicore processors. The optimized SpMV kernel in the CSR format on Intel Xeon Phi can be found in our previous work [72].

4.5.3 Computation of Real-Space Terms

The real-space sum of the PME method is performed by interacting pairs of particles within a short cutoff radius, depending on the Ewald parameter α. Since this operation must be repeated in the BD algorithm, we store the real-space sum operator, M^{real}, as a sparse matrix. To construct this sparse matrix, only the RPY tensors between particles within a short distance need to be evaluated, which we compute efficiently in linear time using Verlet cell lists [4].

This sparse matrix has 3×3 blocks, owing to the tensor nature of the RPY tensor. We thus store the sparse matrix in Block Compressed Sparse Row (BCSR) format. Previously, we optimized SpMV for this matrix format, using thread and cache blocking, as well as code generation to produce fully-unrolled SIMD kernels for SpMV with a block of vectors [71]. The latter is required in Algorithm 8 as multiple time steps are taken with the same mobility matrix, and thus it is possible and efficient to operate on multiple vectors simultaneously.

4.5.4 Performance Modelling and Analysis

The performance of each step of PME is modeled separately. We focus on the reciprocal-space part, since the real-space part is a straightforward sparse matrix-vector product. We also exclude the construction of P from our analysis, since it is a preprocessing step when PME is applied to multiple force vectors with the same particle configuration.

(a) *Spreading*: The total memory traffic, in bytes, incurred by spreading is

$$M_{spreading} = (3 \times 8 \times K^3) + (12 \times p^3 n) + (3 \times 8 \times p^3 n)$$

where the first term is the memory traffic due to the initialization of F^θ, the second term is the memory footprint of P which includes non-zeros and CSR column indices, and the last term represents the memory traffic due to writing the product of $P^T \vec{f}^\theta$ to F^θ. Since the spreading step is performed by SpMV, the performance is bounded by memory bandwidth.

The execution time is estimated as

$$T_{spreading} = \frac{M_{spreading}}{B}$$

where B is the hardware memory bandwidth.

(b) *3D FFTs*: Each PME operation for the RPY tensor requires three forward 3D FFTs and three inverse 3D FFTs with dimensions $K \times K \times K$. We model the execution time as

$$T_{FFT} = 3 \times 2.5K^3 \log_2(K^3)/P_{FFT}(K)$$

$$T_{IFFT} = 3 \times 2.5K^3 \log_2(K^3)/P_{IFFT}(K)$$

where the numerators are the number of flops required for radix-2 3D FFTs, and where the denominators $P_{FFT}(K)$ and $P_{IFFT}(K)$ are the achievable peak flop rates of forward and inverse 3D FFTs, respectively.

(c) *Applying Influence Function*: Our implementation of applying the influence function constructs stores only one word per 3×3 tensor. Thus, the memory traffic due to accessing I is only $8 \times K^3/2$. As discussed in Section 4.5.2, the performance of this step is memory bandwidth bound. Therefore, the execution time can be expressed as

$$T_{influence} = \frac{(8 \times K^3/2) + (2 \times 3 \times 16 \times K^3/2)}{B}$$

where the second term in the numerator is the total memory footprint of C^θ and D^θ.

(d) *Interpolation*: The memory traffic incurred by interpolation is similar to that of spreading except that interpolation does not require initializing U^θ. Its execution time can be expressed as

$$T_{interpolation} = \frac{(12 \times p^3 n) + (3 \times 8 \times p^3 n)}{B}.$$

The overall performance model of the reciprocal-space PME calculation is the sum of the terms above,

$$T_{reciprocal} = \frac{7.5K^3 \log_2(K^3)}{P_{FFT}(K)} + \frac{7.5K^3 \log_2(K^3)}{P_{IFFT}(K)} + \frac{72p^3 n + 76K^3}{B}$$

The memory requirement in bytes for this part of PME can be expressed as

$$M_{reciprocal} = (3 \times 8 \times K^3) + (12 \times p^3 n) + (8 \times K^3/2)$$

where the first term is the storage for F^θ and U^θ (or C^θ and D^θ), the second term represents the memory footprint of P, and the last term is the storage for the influence function. Since the number of mesh points, K^3, is generally chosen to be proportional to the number of particles n (assuming fixed volume fraction), the reciprocal-space part of PME scales as $O(n \log n)$ and requires $O(n)$ storage.

(e) *Computation of Real-space Sum*: In previous work [71], we developed a performance model for SpMV in Stokesian dynamics simulations, which is also applied to the performance of $M^{real} f$. With a small cutoff distance r_{max}, the number of nonzeros per row of M^{real} is much less than the number of particles. Thus, the real-space calculation has a complexity of $O(n)$ and requires $O(n)$ storage.

In order to use that model, we need to estimate the number of nonzero blocks of $(M^{real} + M^{self})$, or the total number of particle interactions within the cutoff radius r_{max},

$$nnzb = \sum_{i}^{n} \frac{4/3\ \pi\ r_{max}^3 \Phi}{4/3\ \pi\ a_i^3}$$

where Φ is the volume fraction, and a_i is the radius of the i-th particle.

4.5.5 Hybrid Implementation on Intel Xeon Phi

With the advance of acceleration hardware such as GPUs and Intel Xeon Phi, it is important to study the coupling of general-purpose processors with accelerators to solve various computational problems. The matrix-free BD algorithm is designed to have low memory requirements, making it possible to use use accelerators with low memory capacities for large-scale BD simulations. In this section, we describe a hybrid implementation of the matrix-free BD method using CPUs and Intel Xeon Phi.

In the PME method, the real-space terms and the reciprocal-space terms can be computed concurrently. It is natural to offload one of these for computation on accelerators.

It is preferable to offload the reciprocal-space calculation for the following reasons. First, the reciprocal-space calculation mainly consists of FFTs, matrix-vector products and the SpMV operations, which are very suitable to be computed on acceleration hardware that has wide SIMD instructions and high memory bandwidth. What is more important that, the bandwidth requirements between CPUs and accelerators through PCI-e for performing the reciprocal-space calculations is relative low because each application only requires communicating two vectors: one vector for the input forces and the other vector for the results. This suggests that offloading the reciprocal-space calculations to computing on accelerators is likely to achieve high performance.

To balance the workload between CPUs and Intel Xeon Phi, the Ewald parameter α is tuned so that one real-space calculation on the CPU and one reciprocal-space calculation on the accelerator consume approximately equal amounts of execution time. To predict the execution time, we use the performance models presented in Section 4.5.4. This works for computing the PME operation in line 9 of Algorithm 2.

The PME operation shown in line 6, however, involves a block of vectors. Here, the real-space part is performed very efficiently with SpMV on a block of vectors. There is no library function, however, for 3D FFTs for blocks of vectors. The reciprocal-space part thus effectively has higher workload when PME is applied on a block of vectors. (The Ewald parameter α may be increased to balance the workload for this case, but practically α is limited if sparsity and scalable storage is to be maintained for the real-space part.) The solution we adopt for the PME operation in line 6 of Algorithm 2 is to also assign to CPUs some reciprocal-space calculations. A static partitioning of the reciprocal-space calculations is performed to achieve load balance between CPUs and multiple Xeon Phi coprocessors.

4.6 Experimental Results

4.6.1 Experimental Setup

Our experimental test-bed is a dual socket Intel Xeon X5680 (Westmere-EP) system with two Intel Xeon Phi coprocessors (KNC) mounted on PCI-e slots. The key architectural parameters are listed in Table 15.

Table 15: Architectural parameters of systems used in performance evaluation.

	2X Intel X5680	Intel Xeon Phi
Microarchitecture	Westmere-EP	MIC
Frequency (GHz)	3.33	1.09
Sockets/Cores/Threads	2/12/24	1/61/244
L1/L2/L3 cache (KB)	64/256/12288	32/512/-
SIMD width (DP, SP)	2-way, 4-way	8-way, 16-way
GFlop/s (DP, SP)	160, 320	1074, 2148
Memory (GB)	24	8
STREAM bandwidth (GB/s)	44	150

Our BD implementations were compiled with Intel ICC 14.0. Intel MKL 11.0 was used to optimize the FFT, BLAS and LAPACK operations in the implementations, including DGEMM, DGEMV, Cholesky factorization, and forward/inverse FFTs.

A monodisperse suspension model of n particles with various volume fractions was used to evaluate the accuracy and the performance of the BD algorithms. For simplicity, van der Waals or electrostatic interactions were not included in the model. To prevent particle overlap, a repulsive harmonic potential between particles was used. The repulsion force between particles i and j both with radius a is given by

$$\vec{f}_{ij}^{repl} = \begin{cases} 125(\|\vec{r}_{ij}\| - 2a)\hat{r}_{ij} & \text{if } \|\vec{r}_{ij}\| \leq 2a \\ 0 & \text{if } \|\vec{r}_{ij}\| > 2a \end{cases}$$

where $\vec{r}_{ij}$ is the vector between particle i and j, and $\hat{r}_{ij}$ is the normalized vector. The repulsion forces were efficiently evaluated using Verlet cell lists [4].

In BD simulations, the translational diffusion coefficients of particles can be estimated by

$$D(\tau) = \frac{1}{6\tau}\left\langle((\vec{r}(t+\tau) - \vec{r}(t))^2\right\rangle \tag{23}$$

where the angle brackets indicate an average over configurations separated by a time interval τ and $\vec{r}$ is the position vector of the particles. For a given BD algorithm, its accuracy can be evaluated by comparing the diffusion coefficients obtained from simulation with theoretical values, values obtained from experiments, or simply values from a known, separately validated simulation.

4.6.2 Accuracy of the Matrix-Free BD Algorithm

For a given α, the accuracy of the PME calculation is controlled by the cutoff distance r_{max}, the mesh dimension K, and the B-spline order p. Using larger r_{max}, K and/or p gives a more accurate result with a more expensive calculation. We measure the relative error of PME as

$$e_p = \frac{\|\vec{u}^{pme} - \vec{u}^{exact}\|_2}{\|\vec{u}^{exact}\|_2}$$

where $\vec{u}^{pme}$ is the result of PME, and $\vec{u}^{exact}$ is a result computed with very high accuracy, possibly by a different method.

The accuracy of the simulation also depends on the accuracy of the Krylov subspace iterations for computing the Brownian displacements. We denote by e_k the relative error tolerance used to stop the iterations.

Parameters for PME and the Krylov subspace method must be chosen to balance computational cost and accuracy. In order to choose these parameters, we performed simulations with different sets of parameters and evaluated the resulting accuracy of the diffusion coefficients obtained from these simulations. Table 16 shows the results. We see that the matrix-free algorithm with $e_k = 10^{-6}$ and $e_p \sim 10^{-6}$ ($e_p \sim 10^{-k}$ means we used parameters giving measured PME relative error between $10^{-(k+1)}$ and 10^{-k}) has a relative error less than 0.25%. The simulations with larger e_k and e_p also achieve good accuracy. Even with

$e_k = 10^{-2}$ and $e_p \sim 10^{-3}$, the average relative error is still lower than 3%. Using larger e_k and e_p significantly reduces running time. The simulations with $e_k = 10^{-2}$ and $e_p \sim 10^{-3}$ are more than 8x faster than those with $e_k = 10^{-6}$ and $e_p \sim 10^{-6}$.

Table 16: Errors (%) in diffusion coefficients obtained from simulations using the matrix-free BD algorithm with various Krylov tolerances (e_k) and various PME parameters (giving PME relative error e_p). Also shown is the execution time (seconds) per simulation step using 2 Xeon CPUs. Simulated systems were particle suspensions of 1,000 particles for various volume fractions Φ.

	$e_k = 10^{-6}$ $e_p \sim 10^{-6}$		$e_k = 10^{-2}$ $e_p \sim 10^{-6}$		$e_k = 10^{-6}$ $e_p \sim 10^{-3}$		$e_k = 10^{-2}$ $e_p \sim 10^{-3}$	
Φ	Error	Time	Error	Time	Error	Time	Error	Time
0.1	0.06	0.089	0.28	0.029	0.42	0.036	0.65	0.010
0.2	-0.09	0.102	-0.22	0.032	0.56	0.047	1.48	0.011
0.3	-0.21	0.116	-0.37	0.032	2.05	0.054	2.38	0.012
0.4	0.25	0.130	0.46	0.036	1.28	0.061	3.72	0.013
0.5	0.16	0.130	0.62	0.038	-3.69	0.062	4.27	0.013

As an example of a BD calculation, Figure 9 shows the diffusion coefficients obtained from matrix-free BD simulations of 5000 particles and various volume fractions. Simulations were performed on our test-bed (using hybrid CPU-accelerator computations) for 500,000 steps with $\lambda_{RPY} = 16$, $e_k = 10^{-2}$ and $e_p \sim 10^{-3}$, taking a total of 10 hours. The diffusion coefficients obtained from the simulations are in good agreement with theoretical values. Qualitatively, the diffusion coefficients are smaller for systems with higher volume fractions (more crowded conditions). Such long simulations also illustrate the importance of reducing wallclock time per timestep, as well as enabling larger simulations.

4.6.3 Simulation Configurations

Table 17 shows the simulation configurations used in our experiments. For each configuration, the PME parameters were chosen such that execution time is minimized while keeping the PME relative error e_p less than 10^{-3}. (The procedure for choosing these parameters is beyond the scope of this paper.) The Krylov convergence tolerance $e_k = 10^{-2}$ was used in

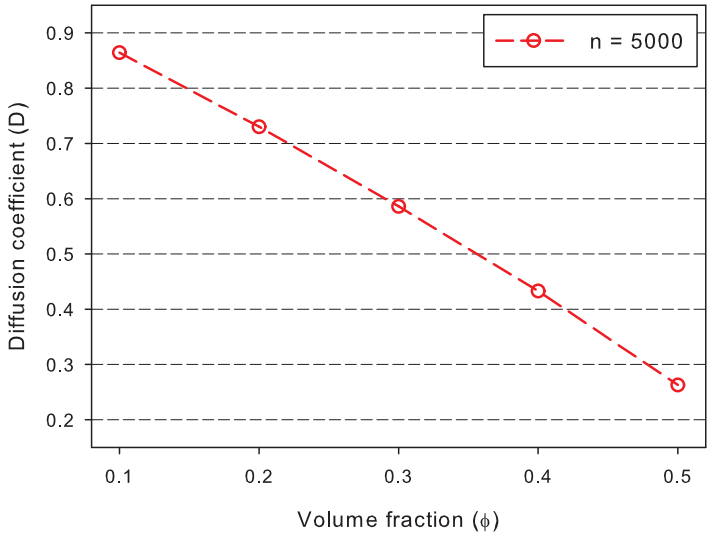


Figure 9: Diffusion coefficients (D) obtained from the simulations using the matrix-free algorithm on a configuration of 5,000 particles with various volume fractions.

all the experiments. In a separate study, we found that if the mesh spacing L/K is fixed, then the reciprocal-space error is independent of the volume fraction. Also, the real-space error only weakly depends on the volume fraction. Therefore, for studying performance, we use a single volume fraction, which we have chosen to be 0.2.

4.6.4 Performance of PME

In matrix-free BD, the same PME operator is applied to different vectors. This is a significant difference from molecular dynamics (or non-Brownian simulations) where a given PME operator is only applied once. Thus an important optimization in our BD case is the precomputation and reuse of the interpolation matrix P. To study the effect of this optimization, we compare the performance of simulations using precomputed P and the application of P on-the-fly (in the latter, P is not stored). The application of P on-the-fly has the advantage of lower memory bandwidth requirements since the elements of P are computed using only the particle positions.

Figure 10 reports this comparison using a CPU-only implementation. In both cases, we used $\lambda_{RPY} = 16$. The number of Krylov subspace iterations for the configurations shown

Table 17: Simulation configurations, where n is the number of particles, K is the PME FFT mesh dimension, p is the B-spline order, r_{max} is the cutoff distance used for the real-space part, α is the Ewald parameter, and e_p is the PME relative error.

Configurations	n	K	p	r_{max}	α	e_p
N100	100	32	4	6.0	0.58	7.05×10^{-4}
N500	500	32	6	6.0	0.58	9.88×10^{-4}
N1000	1,000	32	6	7.5	0.46	6.70×10^{-4}
N2000	2,000	64	6	6.0	0.58	5.88×10^{-4}
N3000	3,000	64	6	6.0	0.58	4.88×10^{-4}
N4000	4,000	64	6	6.0	0.58	8.31×10^{-4}
N5000	5,000	64	6	6.5	0.52	6.82×10^{-4}
N6000	6,000	64	6	6.5	0.52	7.92×10^{-4}
N7000	7,000	64	6	6.5	0.52	8.84×10^{-4}
N8000	8,000	64	6	7.0	0.50	8.81×10^{-4}
N10000	10,000	64	6	7.5	0.46	5.44×10^{-4}
N20000	20,000	128	6	6.0	0.58	3.24×10^{-4}
N30000	30,000	128	6	6.0	0.58	4.15×10^{-4}
N50000	50,000	128	6	6.0	0.58	9.90×10^{-4}
N80000	80,000	128	6	7.0	0.50	9.47×10^{-4}
N200000	200,000	256	4	6.0	0.58	9.86×10^{-4}
N300000	300,000	256	6	6.0	0.58	5.82×10^{-4}
N500000	500,000	256	6	7.0	0.50	3.39×10^{-4}

in the figure varies between 19 and 25, meaning a precomputation of P will be reused more than 300 times. The results show that precomputing P gives on average 1.5x speedup compared to using the on-the-fly implementation. The largest speedup is achieved by the configurations with larger values of p^3n/K^3 (*N10000*, *N80000*, *N500000*). This is because the complexity of computing P is a function of p^3n and the computational cost of the other steps is mainly dependent on K^3.

The overall performance of the reciprocal-space part of PME is shown in Figure 11 as a function of the number of particles and of the PME mesh dimension. The break down of the time for each phase is also shown. Timings are for the CPU-only implementation. The main observation is that FFT operations generally dominate the execution time. However, the execution time of spreading and interpolation operations increases rapidly with the number of particles. These operations are memory bandwidth limited and are very costly

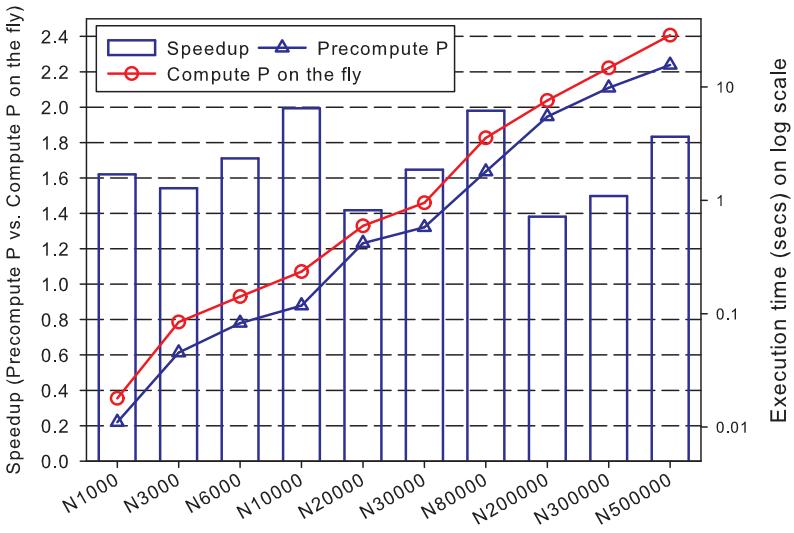
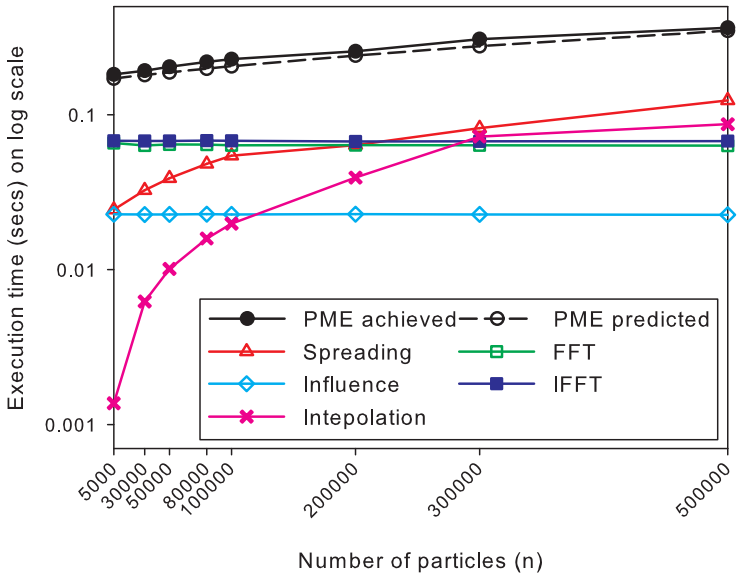


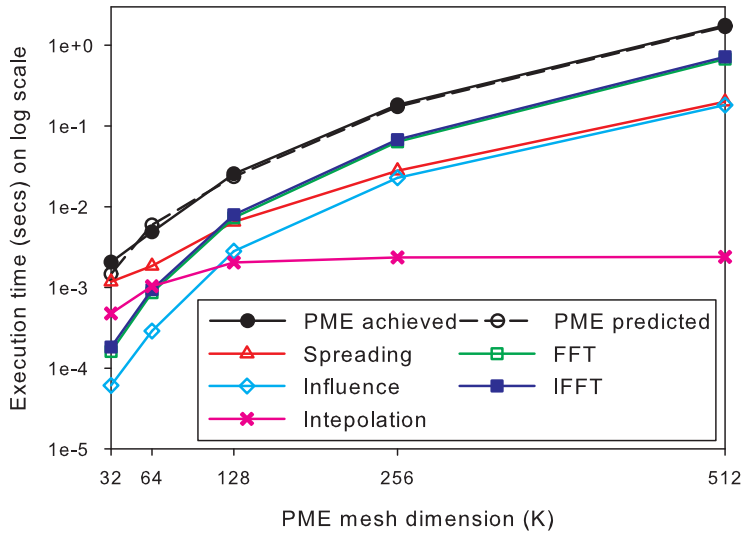
Figure 10: Performance comparison of the PME implementation (reciprocal-space part only) that precomputes P with the implementation that computes P on-the-fly.

for large numbers of particles, and can surpass the cost of the FFTs. We also observe that applying the influence function, although it is embarrassingly parallel, also becomes costly for large mesh dimensions. This is due to the high bandwidth requirements of applying the influence function. Finally, the figures also show that the achieved performance closely matches the predicted performance from the performance model, indicating that our CPU-only PME implementation is as efficient as possible.

In Figure 12 we compare the performance of the reciprocal-space part of PME on two different architectures: Westmere-EP and KNC in native mode. For small numbers of particles, KNC is only slightly faster than or even slower than Westmere-EP. This is mainly due to inefficient FFT implementations in MKL on KNC, particularly for the 3D inverse FFT (this is currently being resolved by Intel). For large numbers of particles, KNC is as much as 1.6x faster. For some unknown reasons, on KNC the inverse 3D FFT routine in MKL only achieves half of the performance of the forward 3D FFT routine on KNC. For reader's reference, the achieved performance of MKL FFT routines is listed in Table 18.



(a) $K = 256, p = 6, r_{max} = 5$



(b) $n = 5000, p = 6, r_{max} = 5$

Figure 11: The overall performance of the reciprocal-space part of PME and the break down of execution time for each phase as a function of the number of particles and the PME mesh dimension.

4.6.5 Performance of BD Simulations

Now we present the overall performance of simulations using the Ewald BD algorithm and the matrix-free algorithm. Figure 13 shows the results. The methods used parameters

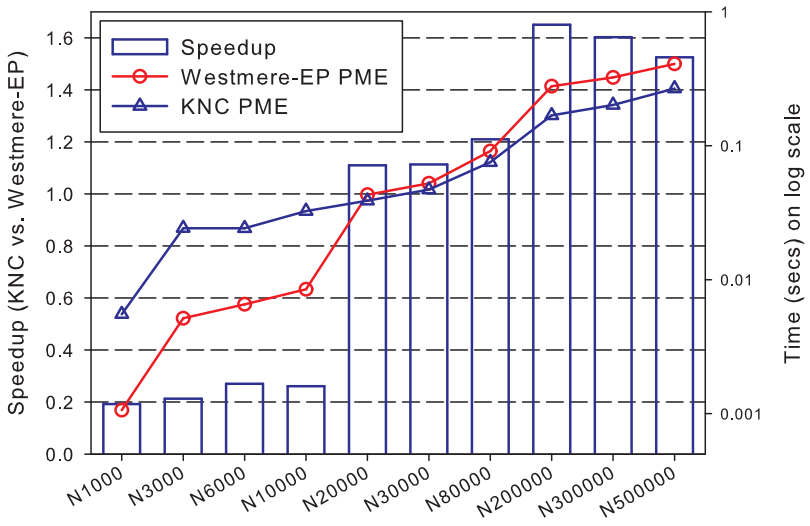
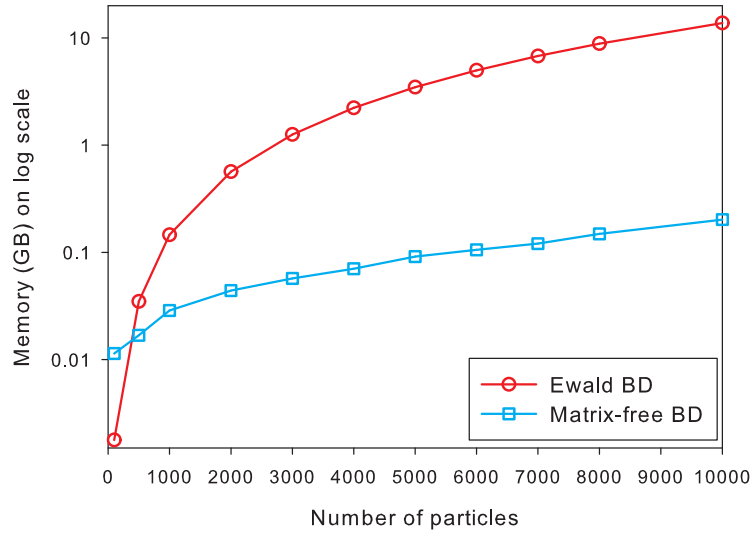


Figure 12: Performance comparison of PME on Westmere-EP with PME on KNC in native mode.

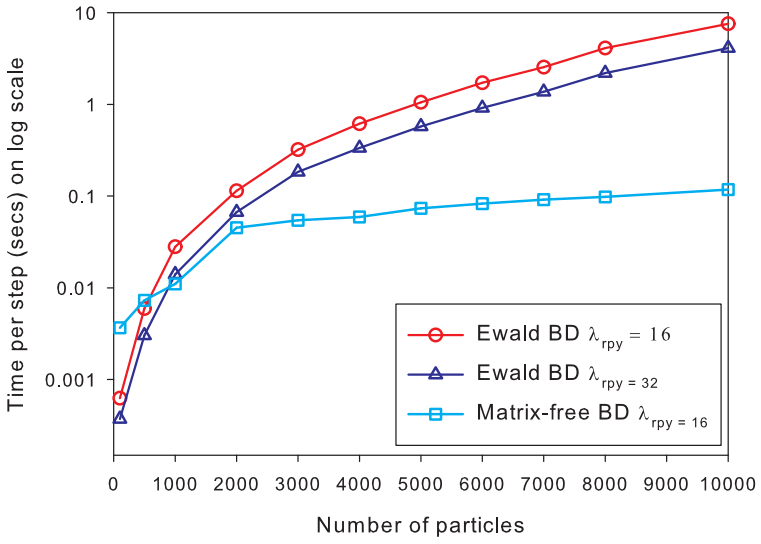
Table 18: Execution time (seconds) of the forward and the inverse 3D FFT routines of MKL on Westmere-EP and KNC.

	forward 3D FFT (GFlops)		inverse 3D FFT (GFlops)	
K	Westmere-EP	KNC	Westmere-EP	KNC
32	23.94	1.55	20.37	1.42
64	42.38	11.04	37.89	9.92
128	45.91	53.13	41.91	36.93
256	47.70	73.56	44.60	39.00
512	40.51	56.93	38.08	34.75

giving results of similar accuracy. As expected, the matrix-free algorithm has great advantages over the Ewald algorithm both in the terms of memory usage and execution time. For large configurations, the speedup of the matrix-free algorithm over the Ewald algorithm is more than 35x. (For problems of this size, the standard algorithm can be improved by using Krylov subspace methods rather than Cholesky factorizations for computing Brownian displacements.) However, the more important result is that, as shown in Figure 14, our implementation of the matrix-free algorithm is capable of performing BD simulations for as many as 500,000 particles.



(a) *Memory*



(b) *Execution time*

Figure 13: Comparison of the Ewald BD algorithm with the matrix-free BD algorithm on Westmere-EP as a function of the number of particles.

Figure 15 compares the performance of our hybrid BD implementation using two Intel Xeon Phi coprocessors with the CPU-only implementation. The hybrid implementation is always faster than the CPU-only implementation for all the configurations, achieving on average a speedup of 2.5x. The largest speedup is achieved by very large configurations,

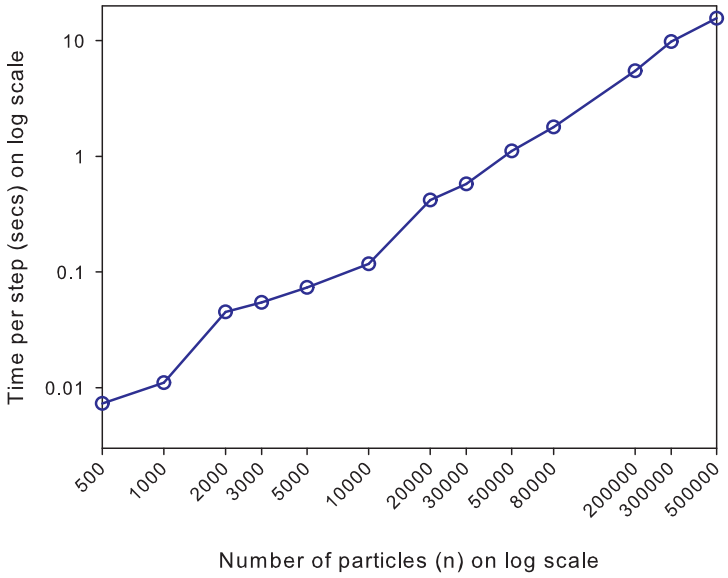


Figure 14: Performance of the matrix-free BD algorithm on Westmere-EP as a function of the number of particles.

which is more than 3.5x. For small configurations, the advantage of the hybrid implementation over the CPU-only implementation is marginal probably because of two reasons. First, the PME implementation on KNC for small configurations is not efficient. Second, for the small configurations there is not enough work to compensate for the communication overhead of offloading.

4.7 Summary

This chapter presented a matrix-free algorithm for Brownian dynamics simulations with hydrodynamic interactions for large-scale systems. The algorithm used the PME method, along with a block Krylov subspace method to compute Brownian displacements in matrix-free fashion. Our software using this algorithm enabled large-scale simulations with as many as 500,000 particles. Our experimental results showed that the new algorithm scales better than the conventional BD algorithm both in the terms of execution time and memory usage.

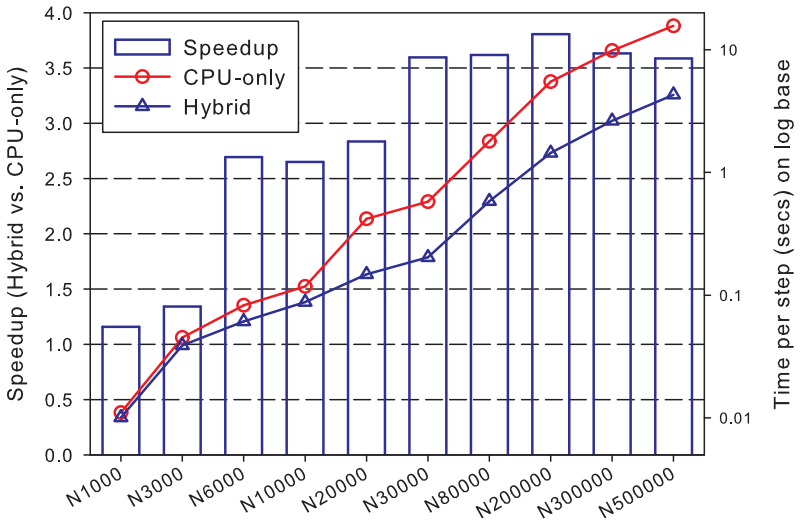


Figure 15: Performance comparison of the hybrid BD implementation with the CPU-only implementation.

We also described an efficient implementation of the matrix-free algorithm on multi-core and manycore processors. The PME algorithm was expressed as the application of a sequence of kernels (SpMV, FFT) to simplify efficient implementation. We also developed a hybrid implementation of BD simulations using multiple Intel Xeon Phi coprocessors, which can be 3.5x faster than the CPU-only implementation.

Chapter V

IMPROVING THE PERFORMANCE OF STOKESIAN DYNAMICS SIMULATIONS VIA MULTIPLE RIGHT-HAND SIDES

This chapter will present an algorithmic approach for improving the performance of scientific applications. The approach is to redesign existing scientific algorithms that use sparse matrix-vector products with a single vector (SpMV) to instead use a more efficient kernel, the generalized SpMV (SPIV), which computes with multiple vectors simultaneously. Given the well-known growing imbalance between memory access and computation rates on modern computer architectures, the computational kernels that operate on multiple vectors are commonly more efficient than those operate on single vectors. This is because the memory bandwidth cost in the formers is amortized over many vectors.

In this chapter, we will take the Stokesian dynamics (SD) method as the example to demonstrate how to exploit SPIV when only one vector is available at a time. However, the proposed approach is very general and can be applied to other molecular simulation applications.

5.1 Motivation

The inspiration of this work is the paper by Gropp et al. [51] in 1999, in which the authors observed that SPIV can be performed in little more time than a traditional SpMV; one could multiply by *four* vectors in about 1.5 times the time needed to multiply by a single vector. Although no specific algorithms were identified in Gropp et al.'s paper about how to take advantage of the small incremental cost of multiplying multiple vectors, there are obvious applications where multiple vectors can be exploited. For example, in a finite element analysis where solutions for multiple load vectors, or more generally, "multiple right-hand

sides" are desired, it is natural to use a block iterative solver, where each iteration involves an SpMV with a block of vectors. Such iterative methods have been avoided because of numerical issues that can arise [86], but these methods can be expected to gain more attention with the increasing performance advantage of SPIV over single-vector SpMV. The importance of block iterative solvers is increasing as well, given the increasing number of applications in which multiple right-hand sides occur, for example, in applications of uncertainty quantification, where solutions for multiple perturbed right-hand sides are desired.

In the above applications, the use of SPIV with a block iterative solver is natural because all the right-hand side vectors are available at the same time. It is not clear, however, how to use SPIV when the right-hand sides are only available sequentially, i.e., one after another. This is the common situation in many types of dynamical simulations where, at each time step, a single right-hand side system is solved, and the solution of this system must be computed before the system at the next time step can be constructed.

In this chapter, we present and test a novel algorithm that can exploit SPIV for Stokesian dynamics simulations, even though the right-hand sides are only available sequentially. The main idea is to set up and solve an auxiliary system of equations with multiple right-hand sides; the solution to this auxiliary system provides good initial guesses for the original systems to be solved iteratively at each time step. Solving the auxiliary system is extra work, but it can be done very efficiently using SPIV, and it is piggy-backed onto a solve which must be performed anyway, leading to an overall reduced computation time. The algorithm can be regarded as an instance of a technique or approach that is applicable to other situations.

In addition to presenting the algorithm above, another purpose of this work is to show experimentally and with some simple analyses the performance advantages of SPIV compared to SpMV. Today, given the increasing gap between memory access and computation rates, we expect that the incremental cost of additional vectors is much smaller. Just like

the fact that flops are becoming "free," additional vectors for SpMV are also becoming free. We hope that this work will encourage exploration into developing algorithms that can use efficient kernels that operate on multiple vectors simultaneously.

5.2 Background: Stokesian Dynamics

5.2.1 Governing Equations

The Langevin equation for particle simulations with Brownian interactions is,

$$m\frac{d^2\vec{r}}{dt^2} = \vec{f}^H + \vec{f}^B + \vec{f}^P$$

where $\vec{r}$ is a $3n$-dimensional vector containing the three components of position of n particles, m is the mass tensor, and $\vec{f}^H$, $\vec{f}^B$, and $\vec{f}^P$ are the hydrodynamic, Brownian, and other external or inter-particle forces, respectively. The Langevin equation is simply a modification of Newton's equation of motion to include the stochastic term $\vec{f}^B$. The equation also often implies that $\vec{r}$ contains a reduced number of degrees of freedom, for instance $\vec{r}$ does not contain degrees of freedom due to the solvent, as the solvent is modeled by $\vec{f}^B$.

In SD simulations, the inertial forces are small and thus particle mass can be neglected, i. e., $md^2\vec{r}/dt^2 = 0$. The hydrodynamic force on a particle is dependent on the positions and velocities of *all* other particles; $\vec{f}^H$ takes the form

$$\vec{f}^H = R\left(\frac{d\vec{r}}{dt} - \vec{u}^\infty\right)$$

where R is a resistance or friction matrix that depends on the particle configuration, and $\vec{u}^\infty$ is the velocity of the bulk flow at the position of the particles. For simplicity, we will use $\vec{u}^\infty = 0$ without loss of generality. We will also assume that there is no other external or inter-particle forces, i. e., $\vec{f}^P = 0$. Expanding all the above in the Langevin equation, the governing equation for SD dynamics is

$$R\frac{d\vec{r}}{dt} = -\vec{f}^B$$

5.2.2 Resistance Matrix

The matrix R, called the *resistance matrix*, describes the relationship between the hydrodynamic forces and the velocities of a system of particles. The exact relationship involves solving the Stokes equations for multiple particles. In the SD method, by contrast, R is constructed by superimposing the analytical solutions for two spherical particles in Stokes flow to approximate the multi-particle solution. Separate analytical solutions are provided for the long- and short-range hydrodynamic interactions, resulting in

$$R = M^{-1} + R_{\text{lub}}$$

with

$$R_{\text{lub}} = R_{\text{2B}} - R_{\text{2B}}^{\infty}$$

Here, M is the dense mobility matrix given by Equation (14), which represents the long-range hydrodynamic interactions, R_{lub} is a sparse matrix representing the short-range lubrication interactions [36]. R_{2B} represents the exact two-body HI, including both far-range and short-range interactions. R_{2B}^{∞} represents two-body, far-range HI. Structurally, R and its two components are block matrices, with blocks of dimension 3×3 [1]. Each block represents the interaction between two particles. The blocks in M are the Oseen or RPY tensors (see Section 4.1); the blocks in R_{lub} are tensors coming from lubrication theory [62, 65]. It is these tensors that make R dependent on the particle positions and radii. In practice, we further adjust R_{lub} to project out the collective motion of pairs of particles [25]. With these choices, M is symmetric positive definite and R_{lub} is symmetric positive semidefinite.

We will see in Section 5.2.4 that each time step of the SD simulations involves solves with the matrix R. For small problems, a Cholesky factorization is used; for the large problems in which we are interested, iterative solution methods have been suggested [110, 105, 115]. Iterative methods involve matrix-vector multiplies with R and, for efficiency, must multiply the dense component of the matrix R using fast algorithms such as particle-mesh

[1] In this work, we do not consider torques and stresslets on particles.

Ewald (PME). Like SPIV, such algorithms may also exploit multiple vectors for efficiency. In this work, we will only study the efficiency of SPIV and leave the study of PME with multiple vectors for future work. We thus use an alternative, sparse approximation to R proposed by [110]

$$R = \mu_F I + R_{\text{lub}}$$

which is applicable when the particle interactions are dominated by lubrication forces. The term $\mu_F I$ is a "far-field effective viscosity" with the parameter μ_F chosen depending on the volume fraction of the particles [110]. We use a slight modification of this technique to account for different particle radii.

5.2.3 Brownian Forces

The Brownian force $\vec{f}^B$ is a Gaussian random vector with mean zero and covariance proportional to R,

$$\langle \vec{f}^B \rangle = 0, \quad \langle \vec{f}^B \, \vec{f}^B \rangle = 2k_B T R \Delta t.$$

where Δt is the time step length, k_B is Boltzmann's constant, and T is the temperature.

Similar to computing Brownian displacements in BD simulations (see Section 4.4.2), the standard method of computing Brownian forces is to use Cholesky factorization, which has computational complexity of $O(n^3)$ and requires $O(n^2)$ storage. In this work, we use a more efficient alternative that computes a shifted Chebyshev polynomial in the matrix R which approximates the square root [42]. Like the Krylov subspace method present in Section 4.4.2, the Chebyshev approach only requires matrix-vector product operations with the matrix R.

5.2.4 SD Algorithm

The governing equation of SD simulations is a differential equation of first order. However, this problem is not smooth due to time-fluctuations in $\vec{f}^B$, and a second-order integrator must be used because of the configuration dependence of R; a first-order integrator makes

a systematic error corresponding to a mean drift, $\nabla \cdot R^{-1}$, see [38, 41, 50]. In practice, an explicit midpoint method can be used, which requires two matrix solves at each time step,

$$\text{Solve } R_k \, \vec{u}_k = -\vec{f}_k^B$$

$$\text{Compute } \vec{r}_{k+1/2} = \vec{r}_k + \tfrac{1}{2}\Delta t \, \vec{u}_k$$

$$\text{Solve } R_{k+1/2} \, \vec{u}_{k+1/2} = -\vec{f}_k^B$$

$$\text{Compute } \vec{r}_{k+1} = \vec{r}_k + \Delta t \, \vec{u}_{k+1/2}$$

where k is a time index and $\vec{u}_k$ represents the velocity vector at time index k. The time step size Δt is chosen such that it is larger than the Brownian relaxation time, but small enough so that particles do not overlap.

Algorithm 9: Conventional SD Algorithm for one time step.

1. Construct $R_k = \mu_F I + R_{\text{lub}}(r_k)$
2. Compute $f_k^B = S(R_k) z_k$
3. Solve $R_k \, u_k = -f_k^B$
4. Compute $r_{k+1/2} = r_k + \frac{1}{2}\Delta t \, u_k$
5. Solve $R_{k+1/2} \, u_{k+1/2} = -f_k^B$
6. Update $r_{k+1} = r_k + \Delta t \, u_{k+1/2}$

A summary of the SD algorithm at each time step is shown in Algorithm 9. In the following, let R_k denote $R(r_k)$, and let z_k denote the standard normal vector generated for step k. $S(R_k)$ represents the shifted Chebyshev polynomial in the matrix R_k. As seen in Algorithm 9, two linear systems with single right-hand side are solved at each time step; it is not obvious how to exploit multiple right-hand sides in SD simulations. In the next section, we will present a new algorithm for SD that exploits multiple right-hand sides.

5.3 Exploiting Multiple Right-Hand Sides in Stokesian Dynamics

The SD method requires the solution of a sequence of related linear systems with matrices R_k which slowly evolve in time as the particles slowly evolve in time. A number of solution techniques for sequences of linear systems can take advantage of the fact that the

matrices are slowly varying. The most obvious technique is to invest in constructing a preconditioner that can be reused for solving with many matrices. As the matrices evolve, the preconditioner is recomputed when the convergence rate has sufficiently degraded. A second technique is to "recycle" components of the Krylov subspace from one solve to the next [90] to reduce the number of iterations required for convergence. A third technique is to use the solution of the previous system as the initial guess for the current system being solved. This is applicable when the solution itself is a slowly varying quantity as the sequence evolves.

At each SD time step, two linear systems must be solved which have the same right-hand sides and which have matrices that are slightly perturbed from each other. The simplest technique for exploiting these properties is to use the solution of the first linear system as the initial guess for the iterative solution of the second linear system.

Different time steps, however, have completely different right-hand sides. As already mentioned, these right-hand sides are in fact random with a multivariate normal distribution. At first glance, it thus does not appear possible that an initial guess is available to aid solving the first linear system of each time step.

We now, in fact, present a way to construct initial guesses for these systems in an efficient way. At two consecutive time steps, k and $k+1$, the linear systems to be solved are

$$\begin{aligned} R_k u_k &= S(R_k) z_k \\ R_{k+1} u_{k+1} &= S(R_{k+1}) z_{k+1} \end{aligned} \tag{24}$$

where an initial guess for the second system is desired. In our approach, instead of solving the first system, the following system, which augments the first system with an additional right-hand side, is solved instead:

$$R_k \left[u_k \; u'_{k+1} \right] = S(R_k) \left[z_k \; z_{k+1} \right] \tag{25}$$

This multiple right-hand side system is solved with a block iterative method. The critical

point is that this solve is expected to cost little more than the solve of the original system with a single right-hand side due to the use of SPIV operations. Since R_{k+1} is close to R_k and $S(R_{k+1})$ is close to $S(R_k)$, the solution u'_{k+1} is an initial guess for the second system (24). The hope is that the number of iterations required to solve the second system is now reduced compared to the extra cost of constructing and solving (25) with the additional right-hand side.

The above procedure is of course extended to as many right-hand sides as is profitable. Thus the solution of one augmented system with m right-hand sides at the beginning of m time steps produces the solution for the first of these time steps and initial guesses for the following $m-1$ time steps. The parameter m may be larger or smaller depending on how R_k evolves and on the incremental cost of SPIV for additional vectors. We refer to m as the *number of right-hand sides*.

A summary of the algorithm for m time steps is shown in Algorithm 10. In the following, we call this algorithm the *Multiple Right-Hand Sides (MRHS) algorithm*. The algorithm requires a vector of initial positions, r_0. Let $U = [u_0, \ldots, u_{m-1}]$ and $Z = [z_0, \ldots, z_{m-1}]$. In step 2, note that a SPIV is also used for constructing the right-hand sides, F^B. We use k to denote the index $0, \ldots, m-1$.

5.4 Generalized Sparse Matrix-Vector Products with Multiple Vectors

In order to fully understand the performance potential of algorithms using multiple vectors (or multiple right-hand sides), we need to better understand the performance of SPIV. In this section we study this performance experimentally and with a simple analytical model. It is important to study optimized implementations of SPIV because it would be these that are used in practice. We use standard performance optimizations for this purpose, but producing a general, highly-optimized implementation of SPIV is outside the scope of this work. In particular, we do not exploit any symmetry in the matrices.

Algorithm 10: MRHS Algorithm for m time steps.

Construct $R_0 = \mu_F I + R_{\text{lub}}(r_0)$
Compute $F^B = S(R_0)Z$
Solve augmented system $R_0 U = F^B$
Compute $r_{1/2} = r_0 + \frac{1}{2}\Delta t\ u_0$
Solve $R_{1/2}\ u_{1/2} = -f_0^B$ using solution u_0 from step 3 as initial guess
Update $r_1 = r_0 + \Delta t\ u_{1/2}$
for $k \leftarrow 1$ **to** $m-1$ **do**
    Construct $R_k = \mu_F I + R_{\text{lub}}(r_k)$
    Compute $f_k^B = S(R_k)z_k$
    Solve $R_k\ u_k = -f_k^B$ using u_k from step 3 as initial guess
    Compute $r_{k+1/2} = r_k + \frac{1}{2}\Delta t\ u_k$
    Solve $R_{k+1/2}\ u_{k+1/2} = -f_k^B$ using solution from step 10 as initial guess
    Update $r_{k+1} = r_k + \Delta t\ u_{k+1/2}$
end

5.4.1 Performance Optimizations for SPIV

5.4.1.1 Single-Node Optimizations

There is a substantial literature exploring numerous optimization techniques for SpMV, i.e., the single-vector case. Vuduc [117] provides a good overview of these techniques. More recently, the performance of various SpMV algorithms has been evaluated by several groups [120, 15]. They cover a wide range of matrices, different storage formats, and various types of hardware. There also exist optimized SPIV implementations. The approaches of these implementations are generally extensions of existing methods used for SpMV or SPMM (sparse matrix-matrix multiply). For instance, Im [57] extended register blocking and cache blocking methods to handle multiple vectors in SPIV. Lee et al. [69] improved SPIV by reducing its memory traffic. The method they used is an extension of the vector blocking method, which was first used in SPMM.

We applied several well-known SpMV optimizations to SPIV, including thread blocking and the use of SIMD. We also implemented TLB and cache blocking optimizations [85]. However, use of large pages made TLB blocking unnecessary for all but unrealistically large number of vectors. We have not used register blocking [57] due to the fact that our

matrices already have natural 3×3 block structure. We store the m vectors in row-major format to take advantage of spatial locality.

While there exist a variety of sparse storage formats, in this work we focus on the widely-used Block Compressed Row Storage (BCRS) format, due to the known block structure of our matrices. Similar to the CSR format, BCRS requires three arrays: an array of non-zero blocks stored row-wise, a column-index array which stores the column index of each non-zero block, and a row pointer array, which stores beginning of each block row.

We have developed a code generator which, for a given number of vectors m, produces a fully-unrolled SIMD kernel, which we call the *basic kernel*. This kernel multiplies a small 3×3 block by a $3 \times m$ block. Multiplication of each matrix element is unrolled by m. The nine elements of a 3×3 block are stored packed in SIMD registers and SIMD shuffle operations are used to extract and replicate required values. It is also possible to use vector blocking for multiple vectors, as this was shown to result in improved register allocation and cache performance [69, 57]. However, for our datasets, increasing m resulted in *at most* a commensurate run-time increase. As a result, vector blocking would not be effective for realistic values of m.

5.4.1.2 Multi-Node Optimizations

Similar to single-node SpMV, multi-node SpMV has been well-studied in the past, for example, see [103, 20]. Our multi-node SPIV implementation is similar to multi-node implementations of SpMV, except that it operates on a block of vectors. For a given matrix partitioning, communication volume scales proportionately with the number of vectors, m.

Strong scaling performance of SPIV is generally limited by two factors: load imbalance and communication overhead. To address load imbalance, we used a simple, coordinate-based row-partitioning scheme. This partitioning bins each particle using a 3D grid and attempts to balance the number of non-zeros in each partition. The entire operation is inexpensive, and can be done during neighbor list construction to further amortize its overhead

over several time steps. Coordinate-based partitioning resulted in communication volume and load balance comparable to that of a METIS [64] partitioning.

To reduce communication overhead, we overlap computation with communication, using nonblocking communication MPI calls. We also overlap the gather of the elements to be communicated with the multiply by the local part of the matrix. We use a small subset of threads to perform the communication and gather operations, while the remaining threads perform the compute.

5.4.2 Performance Model

5.4.2.1 Single-Node Bound

Gropp et al. [51] analyzed the benefits of multiplying a sparse matrix by multiple vectors. However, only the bandwidth-bound case was analyzed. We also analyze the compute-bound case, which can arise for large-enough m. We also slightly extend the performance model to block-structured matrices which arise in many applications including SD simulation.

We now define some quantities used frequently in this paper. For the SPIV operation $Y = RX$, let n denote the number of rows and let n_b denote the number of block rows in the matrix R (for 3×3 blocks, $n_b = n/3$). Further, let n_{nz} denote the number of stored scalar non-zeros in the matrix and let n_{nzb} denote the number of block non-zeros in the matrix. Let s_a be the size, in bytes, of a matrix block (for 3×3 blocks, $s_a = 72$ in double precision). Let s_x be the size, in bytes, of a scalar entry of the (dense) vectors to be multiplied.

The total amount of memory traffic in bytes incurred by a SPIV operation is

$$M_{tr}(m) = mn_b(3 + k(m))s_x + 4n_b + n_{nzb}(4 + s_a)$$

Memory traffic due to non-zeros as well BCRS indexing structures, is represented by the second and third terms in the expression. The first term represents memory traffic due to accessing X and Y: 1 read of X, 1 read of Y, one write to Y, plus $k(m)$ additional memory accesses to each element of X. The function $k(m)$ depends on matrix structure as well as

machine characteristics, such as cache size. The function $k(m)$ also depends on m: as the number of vectors increases, the working set also increases and will put additional pressure on the last level cache. In this case, $k(m)$ will also increase. Cache blocking and matrix reordering techniques will reduce the value of $k(m)$. Note $k(m)$ can also be negative, which can happen for example, when both X and Y fit into last level cache and are retained there between multiple calls to SPIV. For the matrices that are typical in our SD simulation, $k(m)$ is only a weak function of m. For example, for a typical SD matrix with 25 non-zero blocks per block row, $k(m)$ is $\sim$3 for m between 1 and 42.

The time for performing SPIV with m vectors, if the operation is bandwidth-bound, is $T_{bw}(m) = M_{tr}(m)/B$, where B is the achievable machine bandwidth. The time in the compute-bound case is $T_{comp}(m) = f_a m n_{nzb}/F$, where f_a is number of flops required to multiply a block element of R by a block element of X. For example, f_a is 18 for the case of a 3×3 block. The quantity F is the achievable compute-bound performance of the basic kernel.

We approximate the performance of SPIV by the maximum determined by the compute and bandwidth bounds, $T(m) = \max(T_{bw}(m),\ T_{comp}(m))$. The relative time, $r(m)$, is defined as the ratio of the time it takes to multiply by m vectors to the time it takes to multiply by one vector. Hence $r(m) = T(m)/T(1)$. Since we assume that $T(1)$ is bandwidth-bound, $r(m) = T(m)/T_{bw}(1)$. We can divide both numerator and denominator by n_b and note that n_{nzb}/n_b is an average number of non-zero blocks per block row of the matrix. The relative time becomes

$$r(m) = \frac{\max\left[m(3+k(m))s_x + 4 + (n_{nzb}/n_b)(4+s_a),\ m f_a (n_{nzb}/n_b)(B/F)\right]}{(3+k(1))\,s_x + 4 + (n_{nzb}/n_b)(4+s_a)} \tag{26}$$

For small values of m, the relative time is generally determined by the bandwidth bound. For larger m, it is possible that the compute-bound may start dominating the performance. This would be the case for large enough values of byte to flop ratio, B/F, and a large number of blocks per block row, n_{nzb}/n_b, in the matrix. It is also possible that if n_{nzb}/n_b is

too low or $k(m)$ is too high, the bandwidth-bound will continue dominating the performance for all values of m. As an example of this, consider a very large diagonal matrix which does not fit into the last level of cache. Clearly, SPIV is bandwidth-bound in this case for any value of m, since there is no reuse of any vector elements.

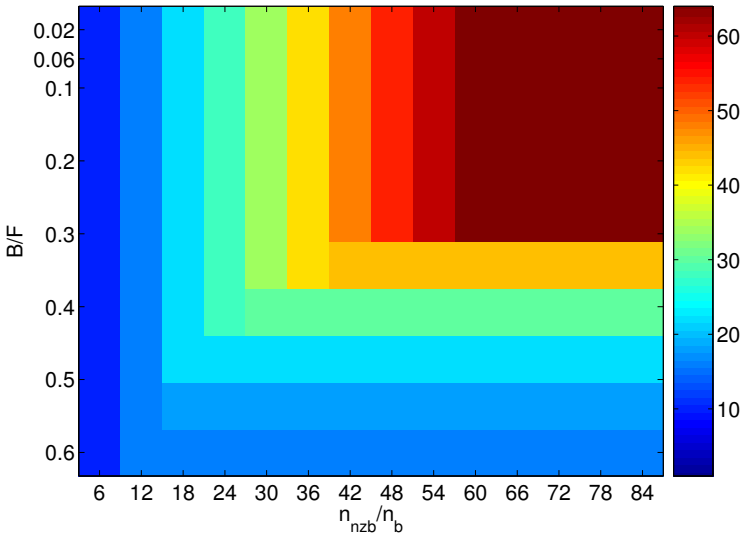


Figure 16: Number of vectors that can be multiplied in 2 times the time needed to multiply by a single vector as a function of n_{nzb}/n_b (x-axis) and B/F (y-axis).

Using the above model, Figure 16 shows a profile of the number of vectors which can be computed in 2 times the time needed to multiply by one vector as n_{nzb}/n_b varies between 6 and 84 and as B/F varies between 0.02 and 0.6. For simplicity, $k(m)$ is optimistically assumed to be 0. The figure shows the trends, but in reality, $k(m)$ is greater than 0 and this will restrict the growth of the number of vectors that can be multiplied in a fixed amount of time. For example, as shown later in Section 5.4.4, for the same values of n_{nzb}/n_b and B/F, the experimentally obtained values of the number of vectors are somewhat smaller than those shown in this profile.

5.4.2.2 Multi-Node Bound

We now extend the definition of relative time to the multi-node case. For p nodes, the relative time $r(m,p)$ is the ratio of time to compute with m vectors on p nodes to the time to compute with a single vector on the same number of nodes. On a single-node

SPIV performance may be bound by bandwidth or computation; on multiple nodes, SPIV performance can be also bound by communication, which will increase with p.

5.4.3 Experimental Setup

In this section we briefly describe the experimental setup for evaluating our SPIV implementations. We introduce relevant hardware characteristics of the evaluated systems and present an overview of the test matrices.

5.4.3.1 Single-Node Systems

We performed single-node experiments on two modern multi-core processors: Intel Xeon Processor X5680, which is based on Intel Core i7, and Intel Xeon Processor E5-2670, which is based on Sandy Bridge. In the rest of the paper, we abbreviate the first architecture as WSM (for Westmere) and the second as SNB (for Sandy Bridge).

WSM is a x86-based multi-core architecture which provides six cores on the same die running at 3.3 GHz. It features a super-scalar out-of-order micro-architecture supporting 2-way hyper-threading. In addition to scalar units, this architecture has 4-wide SIMD units that support a wide range of SIMD instructions called SSE4 [59]. Together, the six cores can deliver a peak performance of 79 Gflop/s of double-precision arithmetic. All cores share a large 12 MB last level L3 cache. The system has three channels of DDR3 memory running at 1333 MHz, which can deliver 32 GB/s of peak bandwidth.

SNB is the latest x86-based architecture. It provides 8 cores on the same die running at 2.6 GHz. It has a 8-wide SIMD instruction set based on AVX [58]. Together, the 8 cores deliver 166 Gflops of double-precision arithmetic. All cores share a large 20 MB last level L3 cache. The system has four channels of DDR3 which can deliver 43 GB/s of peak bandwidth.

We see that compared to WSM, SNB has 2.1 times higher compute throughput but only 1.3 times higher memory bandwidth. Effectively, compared to WSM, SNB can perform 1.6 times more operations per byte of data transferred from memory.

5.4.3.2 Multi-Node Systems

We performed multi-node experiments on a 64-node cluster. Each node consists of a dual-socket CPU with the same configuration as WSM, described in the previous section, except it runs at the lower frequency of 2.9 GHz. The nodes are connected via an InfiniBand interconnect that supports a one-way latency of 1.5 *usecs* for 4 bytes, a uni-directional bandwidth of up to 3380 MB/s and bi-directional bandwidth of up to 6474 MB/s. Note that in our experiments we have only used a single socket on each node.

Table 19: Three matrices from SD.

Matrix	n	n_b	n_{nz}	n_{nzb}	n_{nzb}/n_b
mat1	0.9M	300K	15.3M	1.7M	5.6
mat2	1.2M	395K	81M	9M	24.9
mat3	1.2M	395K	162M	18M	45.3

5.4.3.3 Test Matrices

To study SPIV, we used three matrices generated by our SD simulator, **mat1**, **mat2** and **mat3**. Table 19 summarizes their main characteristics. We changed the cutoff radius in the SD simulator to construct matrices with different values n_{nzb}/n_b.

5.4.4 Experimental Results

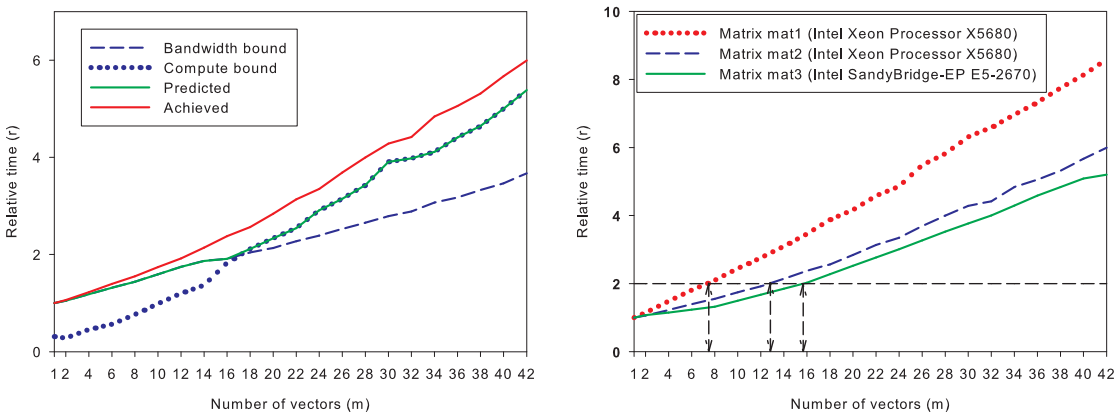
5.4.4.1 Compute and Bandwidth Bounds

The performance model described in Section 5.4.2 requires B/F, which is the ratio of STREAM bandwidth to the achievable floating-point performance of the basic kernel. Running STREAM to obtain B on both architectures shows that WSM achieves 23 GB/s, while SNB achieves 33 GB/s, which is a factor of 1.5 improvement over WSM, due to the additional memory channel. To obtain F, we constructed a simple benchmark that repeatedly computed with the same block of memory. We ran this benchmark for various values of m between 1 and 64. If we exclude $m = 1$, which achieves low performance on both architectures due to low SIMD parallelism, on average this benchmark achieved 45 Gflops

on WSM and 90 Gflops on SNB. The standard deviation from this average is ∼11% for both architectures, the maximum deviation is 13% for WSM and 17% on SNB. The factor of 2 speedup of SNB over WSM is commensurate with their peak floating-point performance ratios. Note also our kernel achieved close to 70% floating-point efficiency on both architectures. The corresponding values of B/F are 0.55 and 0.37 for WSM and SNB, respectively.

Table 20: Performance and bandwidth usage of SpMV ($m = 1$).

	mat1, WSM	**mat2**, WSM	**mat3**, SNB
GB/s	17.8	18.3	32.0
Gflops	3.6	4.2	7.4



(a) (b)

Figure 17: Relative time, r, as a function of m. (a) correlation between performance model and achieved performance for **mat2** on WSM, (b) $r(m)$ for three matrices.

5.4.4.2 Single-Node Results

Table 20 shows performance and bandwidth utilization of single-vector SpMV on both architectures and three matrices. It serves as our baseline. We can see our single vector performance is within 20% of achievable bandwidth on WSM and within 3% on SNB. The reason for such high bandwidth efficiency on SNB is its large 20 MB last level cache

which retains a large part of the X and Y vectors (example of negative $k(m)$ discussed in Section 5.4.2.1). Note that we ran **mat1** and **mat2** matrices on WSM, while to capture the cumulative effects of increased n_{nzb}/n_b and B/F on SPIV performance, we ran **mat3** on SNB.

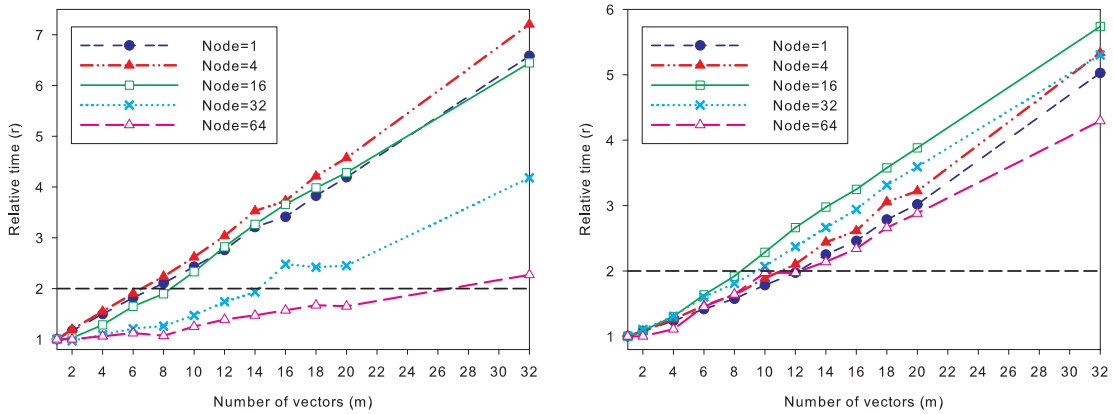
Figure 17(a) shows the achieved (red solid curve) versus predicted (green solid curve) relative time, r, for **mat2** on WSM, as m varies from 1 to 42. As described in Section 5.4.1 achieved performance is the maximum of compute and bandwidth bounds. These two bounds are represented by dotted and dashed curves in the figure. The results show that our predicted relative time closely matches the trend in achieved relative time. A similar match between predicted and achieved relative times was observed for the other two matrices (not shown here for brevity).

Figure 17(b) shows the relative time as a function of m for all three test matrices. The red curve at the top represents the relative time for **mat1** on WSM. We see that for this matrix, we can compute 8 vectors in 2 times the time of a single vector. The is the smallest number of vectors, compared to the other two matrices, when run on the same hardware. This is not surprising because, as Table 19 shows, **mat1** has very small n_{nzb}/n_b. As a result, it is bandwidth-bound for any number of vectors. The blue curve in the middle shows the relative time for matrix **mat2** on WSM. We see that for this matrix, we can multiply as many as 12 vectors in 2 times the time needed to multiply a single vector: 4 more vectors compared to **mat1**. This is due to the fact that **mat2** has larger n_{nzb}/n_b, compared to **mat1**. Finally, the bottom green curve shows relative time for matrix **mat3** on SNB. Note this matrix has the highest n_{nzb}/n_b, compared to the other two matrices, **mat1** and **mat2**. Moreover, SNB has higher B/F, compared to WSM. As a result, we see that in this configuration we can multiply as many as 16 vectors in 2 times the time needed to multiply one vector.

5.4.4.3 *Multi-Node Results*

We describe the performance of SPIV on multiple nodes using two matrices **mat1** and **mat2**. Figure 18 shows the relative time as m varies from 1 to 32 and number of the nodes is increased from 1 to 64. As defined earlier, for a given number of nodes, the *relative time* is the ratio of time required to multiply by m vectors to the time required to multiply by a single vector on the same number of nodes.

For small numbers of nodes, e. g., 4 and 16, the relative time curves are *somewhat higher* but similar to the case for a single node. The slight increase may be attributed to the cost of gathering remote vector values. For large numbers of nodes, e. g., 64, the relative time curves are *lower* than for the single node case. This is because communication costs dominate for the case of large numbers of nodes (as shown in Table 21). Therefore, the additional compute required as the number of vectors increases does not significantly affect the overall time of SPIV. In addition, the communication time of SPIV on large numbers of nodes is mainly consumed by message-passing latency. For a given number of nodes, the time increases very slowly with increasing numbers of vectors. This leads to lower values of relative time for large numbers of nodes.



(a) Matrix **mat1** (b) Matrix **mat2**

Figure 18: Relative time for SPIV using matrix (a) **mat1** and (b) **mat2** as a function of m for various number of nodes up to 64.

Table 21: SPIV communication time fractions for **mat1** matrix. The communication time is significantly higher than the computation time for 32 and 64 nodes. This is not surprising given **mat1**'s low n_{nzb}/n_b of only 5.6.

	Number of vectors, m		
	1	8	32
32 nodes	88%	76%	52%
64 nodes	97%	90%	67%

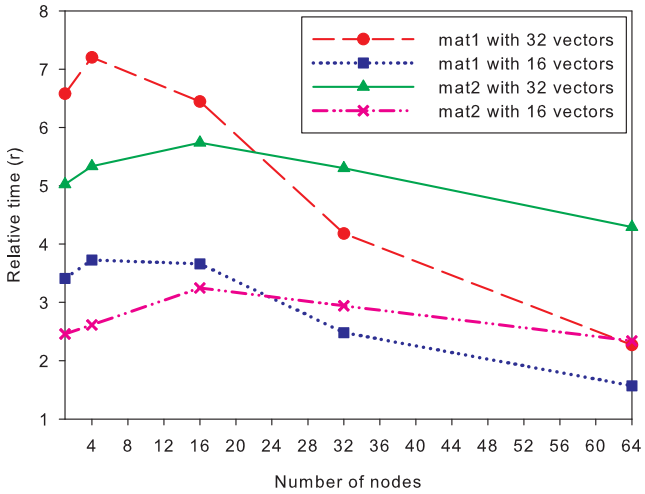


Figure 19: Relative time for SPIV as a function of number of nodes.

In summary, Figure 19 shows the trend in relative time as a function of the number of nodes. As explained above, the relative time increases slightly and then decreases. These results show preliminarily that the use of SPIV is particularly effective when using large numbers of nodes. Further experiments, however, are needed to test other types of matrices and other partitioning schemes, as well as potentially other implementations.

5.5 Stokesian Dynamics Results

In this section, we test the performance of the multiple right-hand side algorithm (Algorithm 10) in a SD application. Indeed, our motivation to improve the performance of SD led to the approach proposed in this paper. Demonstrating the algorithm in the context of an actual application is important because we are then using a sequence of matrices with an

Table 22: Distribution of particle radii.

Particle radius (Å)	Distribution (%)
115.24	2.43
85.23	3.16
66.49	6.55
49.16	0.97
45.43	0.49
43.06	3.64
42.48	2.91
39.16	2.67
36.76	8.01
35.94	8.01
31.71	10.92
27.77	25.97
25.75	8.25
24.01	9.95
21.42	6.07

application-determined variation, rather than an artificial sequence of matrices which may be parameterized to vary faster or slower.

5.5.1 Simulation Setup

Our test system is a collection of 300,000 spheres of various radii in a simulation box with periodic boundary conditions. The spheres represent proteins in a distribution of sizes that matches the distribution of sizes of proteins in the cytoplasm of *E. coli* [7] (see Table 22). The volume occupancy of molecules in the *E. coli* cytoplasm may be as high as 40 percent. Volume occupancy significantly affects the convergence behavior of the iterative algorithms used in SD. Systems with high volume occupancies tend to have pairs of particles which are extremely close to each other, resulting in ill-conditioning of the resistance matrix. Since convergence behavior is a critical factor in the performance of the MRHS algorithm, we test a range of volume occupancies: 10%, 30% and 50%.

The time step length for the simulations is 2 ps. This is the maximum time step size that can be used while avoiding particle overlaps in the simulation. Use of a smaller time

step decreases the overall simulation rate. For computing the Brownian forces to a given accuracy, we have set the maximum order of the Chebyshev polynomial to 30 (i.e., 30 sparse matrix-vector multiplies to compute the Chebyshev polynomial of a matrix times a vector).

Our SD code was written in standard C99, and was compiled with Intel ICC 11.0 using –O3 optimization. All the experiments were carried out on a dual-socket quad-core (Intel Xeon E5530) server with 12 GB RAM using OpenMP or multicore BLAS parallelization. We do not currently have a distributed memory SD simulation code. Such a code would be very complex, needing new algorithms for parallelization and load balancing which we are also developing. In any case, the performance results for SPIV on shared memory and distributed systems, as was shown in Section 5.4, are qualitatively similar, and thus we expect similar conclusions for distributed memory machines.

5.5.2 Experimental Results

5.5.2.1 Accuracy of the Initial Guesses

As described in Section 5.3, the MRHS algorithm processes chunks of m time steps together. At the beginning of every m time steps, one augmented system with m right-hand sides is solved to provide the solution for the first time step and initial guesses for the following $m-1$ time steps. The effectiveness of these initial guesses depend critically on how quickly the resistance matrix R changes as the time steps progress. To obtain some quantitative insight, Figure 20 shows the norm of the difference between the solution and the initial guess for several time steps. An important observation is that the discrepancy between the initial guesses and the solutions appear to increase as the square root of time. This result is consistent with the fact that the particle configurations due to Brownian motion also diverge as the square root of time. This is a very positive result because it implies that changes in the matrix R with respect to an instance at a given point in time actually *slow down* over time. This suggests the possibility that using a large number of right-hand

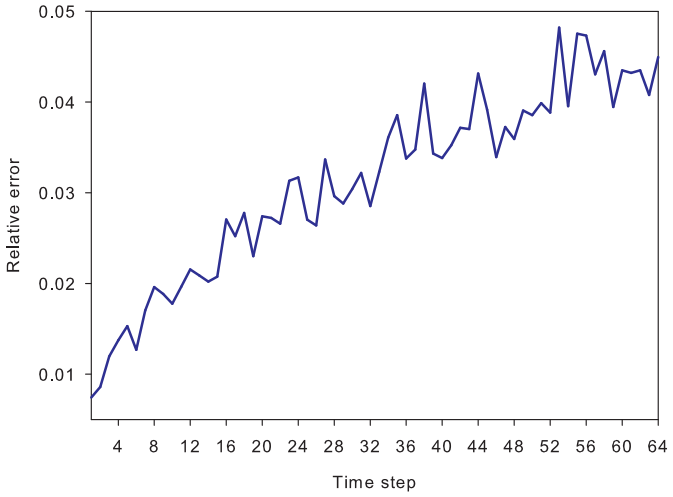


Figure 20: The relative error $||(u_k - u'_k)||_2/||u_k||_2$, where u_k and u'_k are the solution and initial guess at time step k, respectively. The system at the first time step is used for generating the initial guesses. The plot shows a square-root-like behavior which mimics the displacement of a Brownian system over time (the constant of proportionality of relative time divided by the square root of the time step number is approximately 0.006). This result is for a system with 3,000 particles and 50% volume occupancy.

sides may be profitable in the MRHS algorithm.

It is, of course, more relevant to measure the actual number of iterations required for convergence as the number of time steps increases, while using initial guesses constructed using the system at the first time step. The results are shown in Figure 21, where indeed, the number of iterations appear to grow slowly over time. In these tests, the conjugate gradient (CG) method was used and the iterations were stopped when the residual norm became less than 10^{-6} times the norm of the right-hand side.

Table 23 shows the number of iterations required for convergence for particle systems with different volume occupancies. For higher volume fractions, the degradation in performance is faster than for lower volume fractions. The table also shows that the number of iterations is reduced by 30% to 40% when initial guesses are used.

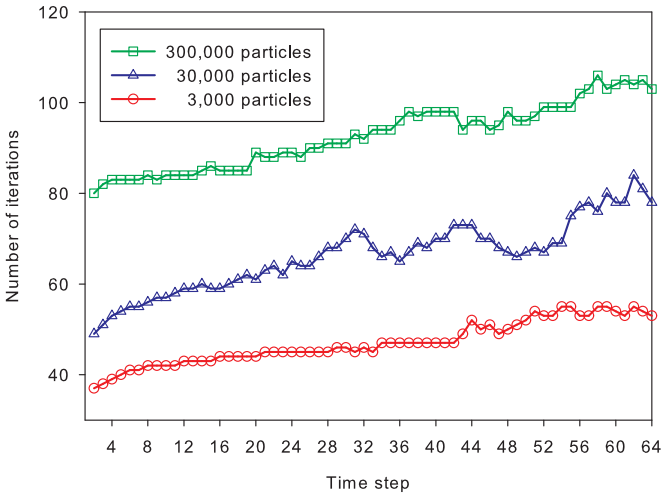


Figure 21: Number of iterations for convergence vs. time step, with initial guesses. The volume occupancy is 50% for the 3 simulation systems.

5.5.2.2 Simulation Timings

We now turn to timings of the SD simulation itself. Tables 25 and 24 show average timings for one time step for SD using the MRHS algorithm and for SD using the original algorithm without initial guesses. The MRHS algorithm used 16 right-hand sides. The tables show the compute time for major components of the simulations. These are: computing Brownian forces with using Chebyshev polynomial approximations using multiple vectors (*Cheb vectors*, step 2 in Algorithm 10); solving the auxiliary system for the initial guesses (*Calc guesses*, step 3 in Algorithm 10), which is only required in the MRHS algorithm; and the two solves with single right-hand sides (*1st solve* and *2nd solve*, steps 10 and 12 in Algorithm 10); as well as Chebyshev with single vector (*Cheb single*, step 9 in Algorithm 10), which are used in both the MRHS algorithm and the original algorithm. Note that in *both* algorithms, in each timestep, the solution of the first solve is used as the initial guess for the second solve.

The results show that, for most cases, the operations for Chebyshev with multiple vectors and the solves with multiple right-hand sides are very efficient. The operations with a block of 16 vectors, for example, are efficient because they are implemented with SPIV. On

Table 23: Number of iterations with and without initial guesses. The table shows the results for 300,000 particle systems with 10%, 30% and 50% volume occupancy.

Step	with guesses			without guesses		
	0.1	0.3	0.5	0.1	0.3	0.5
2	8	12	80	16	30	162
4	8	13	83	16	30	161
6	8	13	83	16	30	162
8	9	14	84	16	30	163
10	9	14	84	16	30	162
12	9	14	84	16	30	162
14	9	14	85	16	30	163
16	9	14	85	16	30	163
18	9	14	85	16	30	162
20	9	14	89	16	30	163
22	9	14	88	16	30	163
24	9	15	89	16	30	163

the other hand, very large m can be used due to the slow degradation of convergence using these initial guesses. The average simulation time per time step is presented in the last row of Table 24 and Table 25, which show that the simulations with the MRHS algorithm are 10% to 30% faster than those with the original algorithm.

5.5.2.3 Analytic Model and Discussions

An important question for the MRHS algorithm is how many right-hand sides should be used to minimize the average time for one simulation step. It can be shown that the best performance is achieved roughly when SPIV switches from being bandwidth-bound to being compute-bound. The details are as follows.

As seen in Figure 21 and Table 23, the number of iterations increases slowly over time. We assume it is constant in the following analysis. Let N denote the number of iterations for the 1st solve without an initial guess. Let N_1 and N_2 denote that number for the 1st solve and the 2nd solve respectively, both with an initial guess. Typically, $N > N_1 > N_2$.

Supposing m right-hand sides are used, the average time for one simulation step with

Table 24: Breakdown of timings (in seconds) for one time step for simulations with varying problem sizes. The volume occupancy of systems is 50%. Note that Chebyshev with multiple vectors and solves with multiple right-hand sides are amortized over many time steps and are not required in the original algorithm (marked by −).

	MRHS algorithm			Original algorithm		
	3000	30000	300000	3000	30000	300000
Cheb vectors	0.025	0.20	1.75	–	–	–
Calc guesses	0.076	0.71	9.66	–	–	–
Cheb single	0.005	0.08	0.84	0.005	0.08	0.84
1st solve	0.007	0.15	2.34	0.014	0.30	4.62
2nd solve	0.003	0.08	1.80	0.004	0.11	2.24
Average	0.021	0.36	5.46	0.023	0.49	7.70

Table 25: Breakdown of timings (in seconds) for one time step for simulations with varying volume occupancy. The results are for systems with 300,000 particles. Note that Chebyshev with multiple vectors and solves with multiple right-hand sides are amortized over many time steps and are not required in the original algorithm (marked by −).

	MRHS algorithm			Original algorithm		
	0.1	0.3	0.5	0.1	0.3	0.5
Cheb vectors	1.09	1.34	1.75	–	–	–
Calc guesses	0.58	1.47	9.66	–	–	–
Cheb single	0.40	0.56	0.84	0.40	0.56	0.84
1st solve	0.12	0.25	2.34	0.22	0.61	4.62
2nd solve	0.08	0.15	1.80	0.08	0.15	2.24
Average	0.66	1.07	5.46	0.70	1.32	7.70

the MRHS algorithm can be expressed as

$$
\begin{aligned}
T_{mrhs}(m) = \frac{1}{m} \Big[& \underbrace{N\,T(m)}_{\textit{Calc guesses}} + \underbrace{C_{max}\,T(m)}_{\textit{Cheb vectors}} \\
& + \underbrace{(m-1)\,N_1\,T(1)}_{\textit{1st solve with an initial guess}} \\
& + \underbrace{m\,N_2\,T(1)}_{\textit{2nd solve}} + \underbrace{(m-1)\,C_{max}\,T(1)}_{\textit{Cheb single}} \Big]
\end{aligned}
\tag{27}
$$

where $T(m)$ is the time for SPIV with m vectors, $T(1)$ is the time for SpMV (with a single vector), and C_{max} is the maximum order of the Chebyshev polynomial. The purpose of our

analysis is to find the value of m which minimizes T_{mrhs}. We denote this value by $m_{optimal}$.

Recall the analysis in Section 5.4, where the performance of SPIV is modeled as $T(m) = \max(T_{bw}(m),\ T_{comp}(m))$. For small values of m, SPIV is bandwidth-bound, where $T(m)$ is equal to $T_{bw}(m)$. As m increases, there are two cases: if $k(m)$ is very large or (n_{nzb}/n_b) is small, the bandwidth bound will continue to dominate, and $T(m)$ is still determined by $T_{bw}(m)$; otherwise, the compute bound starts to dominate, and at some value of m (denoted by m_s), SPIV switches from being bandwidth-bound to being compute-bound. In this case, $T(m)$ is equal to $T_{comp}(m)$.

In our SD simulations, most systems are in the second case, thus $T(1)$ and $T(m)$ can be expressed as

$$
\begin{aligned}
T(1) &= \big(n_b\ (3+k(1))\ s_x + 4n_b + n_{nzb}\ (4+s_a)\big)/B \\
T(m) &= \begin{cases}
\big(m\ n_b\ (3+k(m))\ s_x + 4n_b & \\
\qquad +n_{nzb}\ (4+s_a)\big)/B & \text{if } m < m_s \\
f_a\ m\ n_{nzb}/F & \text{if } m \geq m_s
\end{cases}
\end{aligned}
\tag{28}
$$

Expanding $T(1)$ and $T(m)$ in equation (27), when $m < m_s$, T_{mrhs} can be expressed as a function of $k(m)$ and m

$$
T_{mrhs}(m < m_s) = (3+k(m))P + \frac{1}{m}\,Q + R \tag{29}
$$

where P, Q and R are all constants,

$$
\begin{aligned}
P &= \frac{(N+N_2+C_{max})\ s_x\ n_b}{B} \\
Q &= \frac{N-N_1}{B}\ \big[(4\ n_b + n_{nzb}\ (4+s_a)) \\
&\qquad - (N_1+N_2+C_{max})\ (3+k(1))\ s_x\ n_b\big] \\
R &= \frac{N_1+N_2+C_{max}}{B}\big[n_b\ (3+k(1))\ s_x + 4n_b + n_{nzb}\ (4+s_a)\big]
\end{aligned}
$$

Typically in SD, n_{nzb} is large, and hence $Q > 0$. When $k(m)$ is small and changes very slowly with m, which is our case as mentioned earlier, the expression (29) is a decreasing function of m.

On the other hand, when $m \geq m_s$,

$$T_{mrhs}(m \geq m_s) = S + W - \frac{S}{m} \tag{30}$$

where

$$S = (N_1 + N_2 + C_{max}) \left[n_b \left(3 + k(1)\right) s_x + 4n_b + n_{nzb} \left(4 + s_a\right) \right]$$
$$W = \frac{f_a \, n_{nzb} \left(N + N_2 + C_{max}\right)}{F}$$

which is an increasing function of m (F is almost constant when SPIV is compute-bound).

Putting these expressions together, we conclude that the best simulation performance is achieved when m is near m_s, i.e., when SPIV switches from being bandwidth-bound to being compute-bound.

We evaluate our analysis by running simulations on various test problems. For each simulation, experiments were performed with different numbers of right-hand sides to determine the values of $m_{optimal}$. SPIV was also run on these test problems to determine m_s.

Figure 22 displays the achieved (red solid curve) versus predicted (green solid curve) average simulation time per time step (T_{mrhs}) for a system with 300,000 particles and 50% volume occupancy. The predicted simulation time is the maximum of the bandwidth-bound and compute-bound estimates of T_{mrhs}. As seen in the figure, the achieved T_{mrhs} first decreases as m increases and starts to increase when m is equal to $m_{optimal}$, which matches the trend of the predicted simulation time. Table 26 compares $m_{optimal}$ and m_s for 5 different simulations, showing that they are indeed very close. The slight differences can be explained by the fact that N_1 is actually increasing in our simulations, although very slowly. These results corroborate our analysis.

Finally, we show some results that investigate the speedup of the MRHS algorithm as we increase the number of threads in a shared-memory computation. Figure 23(a) shows the computation time of SPIV for different numbers of threads. For 8 threads, the ratio

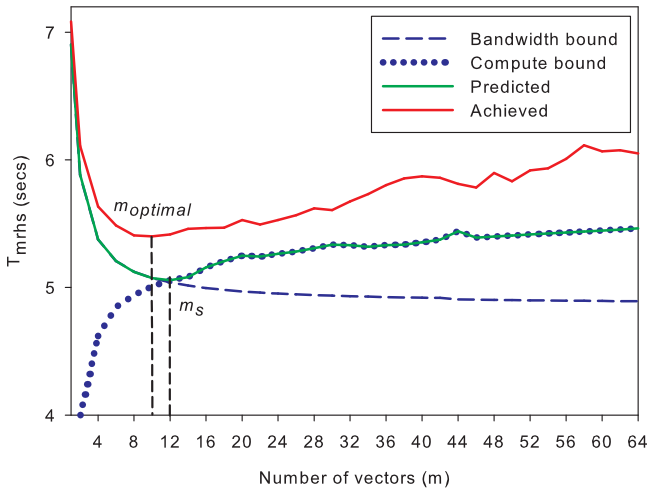


Figure 22: Predicted and achieved average simulation time per time step vs. m. The result is for a system with 300,000 particles and 50% volume occupancy. Equations (29) and (30) were used to calculate the compute-bound and bandwidth-bound estimates with the following parameters: $N = 162$, $N_1 = 80$, $N_2 = 63$, $C_{max} = 30$, $B = 19.4GB/s$ (STREAM bandwidth). F and $k(m)$ are measured values.

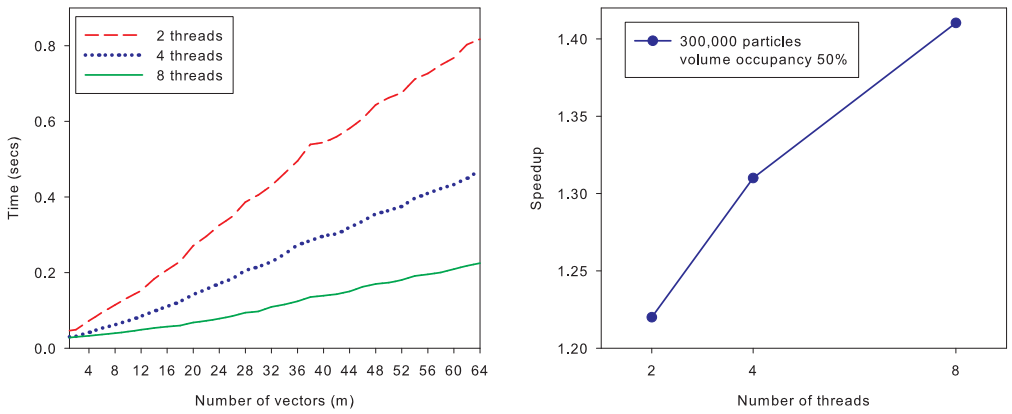
Table 26: m_s and $m_{optimal}$ for different systems.

Problem size	Volume occupancy	m_s	$m_{optimal}$
3,000	50%	5	4
30,000	50%	12	10
300,000	10%	15	12
300,000	30%	13	10
300,000	50%	12	10

B/F is smaller than for 2 or 4 threads. As a result, the speedup with 8 threads shown in Figure 23(b) is larger than that with fewer threads. This result demonstrates the potential of using the MRHS algorithm with large manycore nodes.

5.6 Summary

We have presented an algorithm for improving the performance of Stokesian dynamics simulations. We redesigned the existing algorithm which used SpMV with single vectors to instead use the more efficient SPIV. The main idea of the new algorithm is to solve an auxiliary system with multiple right-hand sides; the solution to this auxiliary system helps



(a) (b)

Figure 23: (a) Performance of SPIV vs. number of threads. (b) Speedup over the original algorithm vs. number of threads. The results are for a system with 300,000 particles and 50% volume occupancy.

solve the original systems by providing good initial guesses. The approach of the algorithm can be extended to other types of dynamical simulations.

We presented a performance model of SPIV and used it to explain SPIV performance. We observed that for matrices with very small numbers of non-zeros per row, SPIV performance is always bandwidth-bound, while for matrices with larger numbers of non-zeros per row, typical for SD and many other applications, SPIV switches from bandwidth-bound to compute-bound behavior with increasing numbers of vectors. In either case, it is typical to be able to multiply a sparse matrix by 8 to 16 vectors simultaneously in only twice the time required to multiply by a single vector. Similar results hold for distributed memory computations. We thus "update" the earlier result reported in [51].

We demonstrated how to exploit multiple right-hand sides in SD simulations. By using the MRHS algorithm, we measured a 30 percent speedup in performance. In addition, we used a simple model to show that the best simulation performance is achieved near the point where SPIV switches from being bandwidth-bound to being compute-bound.

The efficiency of the MRHS algorithm depends on properties of the system being simulated and also characteristics of the hardware. With the ever-increasing gap between

DRAM and processor performance, we expect that the effort of exploiting multiple right-hand sides will become even more profitable in the future. Our work is a good case to study, and we hope it would advocate more algorithm redesigning that exploits multiple right-hand sides.

Chapter VI

EFFICIENT SPARSE MATRIX-VECTOR MULTIPLICATION ON X86-BASED MANY-CORE PROCESSORS

Molecular simulation applications share a number of common computational kernels, such as sparse matrix-vector multiplication (*SpMV*), and fast Fourier transform (*FFT*). The performance of a molecular simulation not only depends on the choice of numerical algorithms, but also on the efficient implementations of these kernels on high-performance hardware. Thus, much research has been devoted to optimize these kernels on a variety of computer architectures.

In this chapter, we will consider the design of efficient SpMV kernels on the Intel Xeon Phi coprocessor (KNC), which a newly released x86-based many-core processor with wide SIMD and a larger number of cores. We expect that the general performance issues raised for SpMV and our solutions also apply to other computational kernels.

6.1 Related Work

Sparse matrix-vector multiplication (SpMV) is an essential kernel in many scientific and engineering applications. It performs the operation $y \leftarrow y + Ax$, where A is a sparse matrix, and x and y are dense vectors. While SpMV is considered one of the most important computational kernels, it usually performs poorly on modern architectures, achieving less than 10% of the peak performance of microprocessors [49, 120]. Achieving higher performance usually requires carefully choosing the sparse matrix storage format and fully utilizing the underlying system architecture.

SpMV optimization has been extensively studied over decades on various architectures. Relevant for us is optimizations for CPUs and GPUs. For a comprehensive review, we refer

to several survey papers [117, 56, 120, 49].

Blocking is widely used for optimizing SpMV on CPUs. Depending on the motivation, block methods can be divided into two major categories. In the first category, blocking improves the spatial and temporal locality of the SpMV kernel by exploiting data reuse at various levels in the memory hierarchy, including register [78, 56], cache [56, 85] and TLB [85]. In the second category, block structures are discovered in order to eliminate integer index overhead, thus reducing the bandwidth requirements [116, 92]. Besides blocking, other techniques have also been proposed to reduce the bandwidth requirements of SpMV. These techniques broadly include matrix reordering [87], value and index compression [119, 67], and exploiting symmetry [21].

Due to the increasing popularity of GPUs, in recent years numerous matrix formats and optimization techniques have been proposed to improve the performance of SpMV on GPUs. Among the matrix formats, ELLPACK and its variants have been shown to be most successful. The first ELLPACK-based format for GPUs was proposed by Bell and Garland [15], which mixed ELLPACK and COO formats. Monakov et al. [83] invented the Sliced ELLPACK format, in which slices of the matrix are packed in ELLPACK format separately, thus reducing zero padding. Vázquez et al. [114] used another approach to address zero-padding. Here, in ELLPACK-R, all the padding zeros are removed, and a separate array is used to store the length of each row. Kreutzer et al. [68] proposed the pJDS matrix format, which extended ELLPACK-R by row sorting. There are also blocked SpMV implementations on GPUs. Jee et al. [24] proposed the BELLPACK matrix format which partitions the matrix into small dense blocks and organizes the blocks in ELLPACK format. Monakov et al. [82] proposed a format also using small dense blocks, but augments it with ELLPACK format for nonzeros outside this structure.

Recently, Su and Keutzer [108] proposed a cross-platform sparse matrix format called cocktail, which combines different matrix formats, and developed a auto-tuning framework

that is able to analyze the sparse matrix at run-time and automatically select the best representations for the given sparse matrix on different architectures.

6.2 Understanding the Performance of SpMV on Intel Xeon Phi

We first evaluated the KNC architecture with a simple SpMV kernel using the widely-used Compressed Sparse (CSR) format. The CSR format is standard, consisting of three arrays: the nonzero values of the sparse matrix are stored in *val*; the column indices corresponding to each nonzero are stored in *colidx*; and the list of indices giving the beginning of each row are stored in *rowptr*.

6.2.1 Test Matrices and Platform

Table 27 lists sparse matrices used in our performance evaluation. These are all the matrices used in previous papers [120, 108, 63] that are larger than the 30 MB aggregate L2 cache of KNC (using 60 cores). A dense matrix stored in sparse format is also included. These matrices come from a wide variety of applications with different sparsity characteristics. No orderings were applied to these matrices, but our results may be better if we use orderings that promote locality when accessing the vector x.

The platform used for experimental tests in this work is a pre-production part of KNC with codename ES B0, which is installed with Intel MIC Platform Software Stack (MPSS) Gold 2.1. The pre-production system is equipped with 8 GB GDDR5 memory and includes a 61-core KNC coprocessor running at 1.09 GHz, which is capable of delivering 1.05 Tflops double precision peak performance. In this work, we use 60 cores to test the SpMV implementations, leaving the remaining core to run the operating system and administrative software. The STREAM Triad benchmark on this system achieves a score of 163 GB/s with ECC turned on. A detailed overview of the KNC Architecture is given in Appendix A.

Table 27: Sparse matrices used in performance evaluation.

Name	Dimensions	nnz	nnz/row
Dense8	8K×8K	64.0M	8000.00
Wind Tunnel	218K×218K	11.6M	53.39
Circuit5M	5.56M×5.56M	59.5M	10.71
Rail4284	4K×1.09M	11.3M	2633.99
Mip1	66K×66K	10.4M	155.77
In-2004	1.38M×1.38M	16.9M	12.23
Si41Ge41H72	186K×186K	15.0M	80.86
Ldoor	952K×952K	46.5M	48.86
Bone010	987K×987K	71.7M	72.63
Rucci1	1.98M×110K	7.8M	3.94
Cage15	5.16M×5.16M	99.2M	19.24
Rajat31	4.69M×4.69M	20.3M	4.33
12month1	12K×873K	22.6M	1814.19
Spal_004	10K×322K	46.1M	4524.96
Crankseg_2	64K×64K	14.1M	221.64
Torso1	116K×116K	8.5M	73.32

6.2.2 Overview of CSR Kernel

Similar to earlier work on SpMV for multi-core CPUs [120, 49], our CSR kernel is statically load balanced by row decomposition, in which the sparse matrix is partitioned into row blocks with approximately equal numbers of nonzeros. Each thread multiplies one row block with a SIMD kernel. We also applied common optimizations to our CSR kernel, including loop unrolling and software prefetching of nonzero values and column indices.

Algorithm 11 shows the pseudocode of the SIMD kernel, in which the inner loop (lines 7-14) processes a matrix row. In each iteration, at most 8 nonzeros are multiplied as follows. First, the elements from *colidx* and *val* are loaded into vectors (lines 10 and 11). Next, the elements from vector *x* are gathered into *vec_x* (line 12). Finally, *vec_vals* and *vec_x* are multiplied and added to *vec_y* by the fused multiply-add instruction (line 13). When fewer than 8 nonzeros are left in a row, only a portion of the slots in the SIMD instructions will be used. In this case, we only update the corresponding elements in the destination vectors. To achieve this, a write-mask (line 9) is used to mask the results of the SIMD instructions.

Algorithm 11: Multiply a row block (from row startrow to endrow) in CSR format (rowptr, colidx, val).

```
1  rowid ← startrow;
2  start ← rowptr [rowid];
3  while rowid < endrow do
4      idx ← start;
5      end ← rowptr [rowid + 1];
6      vec_y ← 0;
       /* compute for a row */
7      while idx < end do
8          rem ← end − idx;
9          writemask ← (rem > 8 ? 0xff : (0xff ≫ (8 − rem)));
10         vec_idx ← load (&colidx [idx],writemask);
11         vec_vals ← load (&val [idx], writemask);
12         vec_x ← gather (vec_idx, &x [0], writemask);
13         vec_y ← fmadd (vec_vals, vec_x, vec_y, writemask);
14         idx ← idx + 8;
15     y [rowid] ← y [rowid] + reduce_add (vec_y);
16     start ← end;
17     rowid ← rowid + 1;
```

The variable *vec_y* is a temporary vector that stores values of *y*. When a row is finished, we perform a vector-wise sum reduction on *vec_y* to get the value of *y* of the row (line 13).

6.2.3 Performance Bounds

We use four performance bounds (measured in flops) to evaluate the performance of our CSR kernel. First, the *memory bandwidth bound performance* is the expected performance when the kernel is bounded by memory bandwidth. It is defined as

$$P_{bw} = \frac{2\,nnz}{M/B}$$

where *nnz* is the number of nonzeros in the matrix, *B* is the STREAM bandwidth of KNC, and *M* is the amount of memory transferred in the SpMV kernel, which is measured by Intel VTune.

The *compute bound performance* is measured by running a modified CSR kernel, in which all memory accesses are eliminated. Concretely, the modified kernel only uses the

first cache lines of *val* and *colidx*. It also eliminates the memory accesses on vector x by setting *colidx* to be $\{0, 8, 16, 24, 32, 40, 48, 56\}$.

The *ideal balanced performance* is the achievable performance when SpMV kernels are perfectly balanced, and is computed as

$$P_{balanced} = \frac{2\,nnz}{T_{mean}}$$

where T_{mean} is the average execution time of the threads.

Finally, the *effective bandwidth bound performance* is the best performance one would expect given that (a) the kernel is bandwidth bound; (b) there are no redundant memory accesses to vectors x and y. It is defined as

$$P_{ebw} = \frac{2\,nnz}{M_{min}/B}$$

where M_{min} is the minimum memory traffic of SpMV assuming perfect reuse of vectors x and y.

Prior work has shown that SpMV is memory bandwidth bound on modern architectures [49, 120, 27]. Thus, it is expected that for all matrices the compute bound performance is larger than the memory bandwidth bound performance. The ideal balanced performance represents the achievable peak performance of a balanced SpMV kernel. For an efficient SpMV implementation, it should be close to the memory bandwidth bound performance. Additionally, the difference between the ideal balanced performance and the measured performance may quantify the degree of load imbalance.

6.2.4 Performance Bottlenecks

Figure 24 shows the performance of our CSR kernel on all test matrices and the corresponding performance bounds. By comparing the measured performance of our CSR kernel to these performance bounds, we discover some unexpected performance issues.

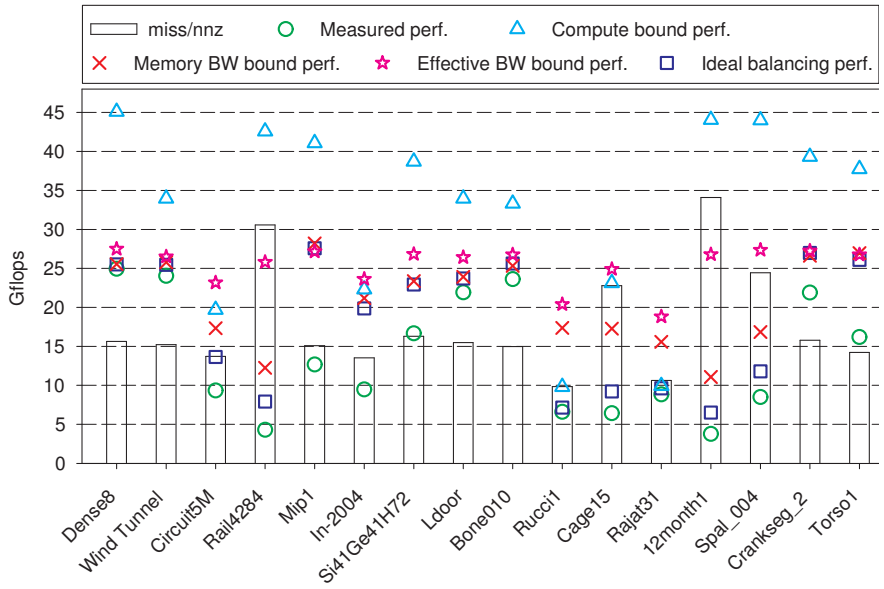


Figure 24: Performance of CSR kernel and performance bounds. The average number of L2 cache misses per nonzero is shown in vertical bar.

6.2.4.1 Low SIMD Efficiency

As seen in Figure 24, for some matrices that have very short row lengths, e. g., *Rajat31* and *Rucci1*, the compute bound performance is lower than the bandwidth bound performance, suggesting that their performance is actually bounded by computation. Previous work on SpMV [49, 23] attributes the poor performance of matrices with short rows to loop overheads. For wide SIMD, we argue that this is also because of the *low SIMD efficiency* of the CSR kernel, i. e., low fraction of useful slots used in SIMD instructions.

More precisely, for a matrix with very few nonzeros per row, the CSR kernel cannot fully utilize KNC's 512-bit SIMD instructions. As a result, the CSR kernel on this matrix performs poorly, and if the SIMD efficiency is very low, the compute bound performance of the kernel may be smaller than the memory bandwidth bound performance. For example, the matrix *Rucci1* has on average 3.94 nonzeros per row. Multiplying the matrix using our CSR kernel on KNC can only use half of the SIMD slots. Thus, the SIMD efficiency is only 50%.

Low SIMD efficiency is not a unique problem for CSR. It can be shown that the compute bound performance will be lower than the memory bandwidth bound performance when the SIMD efficiency is lower than some threshold. Given a matrix format, its SpMV implementation on KNC usually includes a basic sequence of SIMD instructions similar to lines 7-14 in Algorithm 11, which multiplies at most 8 nonzeros (in double precision). Assuming that the basic SIMD sequence takes C cycles (C accounts for both SIMD instructions and loop overheads) and F is the clock speed (Hz), then the compute time for the SIMD sequence is C/F. The minimum memory transfer time for the SIMD sequence is $(8s \times P \times \eta)/B$, where s is the working set size (in bytes) for one nonzero, where P is the number of cores in use, η is the SIMD efficiency, and B is the STREAM bandwidth. To avoid being bounded by computation, the compute time should be greater than the memory transfer time, thus η needs to satisfy

$$\eta > \eta_{min} = \frac{B}{8s \times P} \times C/F \tag{31}$$

This provides the minimum required SIMD efficiency for the given matrix format. We will revisit this in Section 6.3.

6.2.4.2 Cache Miss Latency

Four matrices, *Rail4284*, *Cage15*, *12month1* and *Spal_004*, have ideal balanced performance much lower than both the compute bound performance and the memory bandwidth bound performance. This suggests that, even if the kernel is perfectly balanced, the performance is neither bounded by computation nor bandwidth.

To determine the reason for this, Figure 24 shows the average number of L2 cache misses (measured by Intel VTune) per nonzero for all test matrices. From this figure, we see that those matrices whose ideal balanced performance is lower than the compute and memory bandwidth bound performance have much more L2 cache misses than other matrices (per nonzero). We also notice that their effective bandwidth bound performance is much higher than the memory bandwidth bound performance, which means there are

a large number of redundant memory accesses on vector x. This implies that the poor performance of these matrices may be because of excessive cache misses due to accessing vector x. Unlike accesses to matrix values, accesses to x are not contiguous and can be very irregular, and we do not have an effective way to perform prefetching on x to overcome cache miss latency. While KNC has hardware support for prefetching irregular memory accesses, it does have an overhead that often negates its benefits.

Cache miss latency is not a unique problem for KNC. However, it is more expensive on KNC than on other architectures. To better understand the problem, we compare the cache miss latency (memory latency) and memory bandwidth of KNC with those of CPUs and GPUs. Sandy Bridge, an example of recent multi-core CPUs, has an order of magnitude smaller latency and 5x less memory bandwidth than KNC. While the memory latency of Sandy Bridge still affects performance, the SpMV kernels on it tend to be memory bandwidth bound. High-end GPUs have comparable memory bandwidth and latency to KNC. However, memory latency is usually not a problem for GPUs as they have a large number of warps per streaming multiprocessor (vs. 4 threads per core on KNC) so the memory latency can be completely hidden.

6.2.4.3 Load Imbalance

In the CSR kernel, we statically partition the matrix into blocks with equal numbers of nonzeros. We call this partitioning scheme *static-nnz*. Although *static-nnz* is widely used in previous SpMV implementations on multi-core processors and leads to good performance, it performs poorly on KNC. As shown in Figure 24, for at least 8 matrices the CSR kernel is highly unbalanced, resulting in a significant performance degradation up to 50%.

The poor performance of *static-nnz* in these cases is mainly due to the fact that the sparsity structure of a sparse matrix often varies across different parts of the matrix. In one case, some parts of the matrix may have more rows with shorter row lengths than other parts. The threads assigned to these parts are slower than the other threads because they

are burdened by more loop overheads. In another case, some parts of the matrix may have very different memory access patterns on vector x than others. For these parts of the matrix, accessing vector x may cause more cache misses. Considering that KNC has far more cores than any other x86 processor, it is more likely to have some partitions with very different row lengths or memory access patterns from others. As a result, while all partitions in *static-nnz* have the same number of nonzeros, they can have very different performance. This explains why *static-nnz* performs well on previous CPUs but is not effective on KNC.

In summary, the results using our CSR kernel on KNC reveal that, depending on the matrix, one or more of three bottlenecks can impact performance: (a) low SIMD efficiency due to wide SIMD on KNC; (b) high cache miss latency due to the coherent cache system; (c) load imbalance because of the large number of cores. We will first address the low SIMD efficiency and the cache miss latency bottlenecks in the next section by proposing a new matrix format. The subsequent section addresses load imbalance.

6.3 Ellpack Sparse Block Format

6.3.1 Motivation

From the architectural point of view, KNC bears many similarities with x86-based multi-core CPUs. On the other hand, there are also similarities between KNC and GPUs, for example, both have wide SIMD. Thus, before considering a new matrix format for KNC, it is necessary to first evaluate existing matrix formats and optimization techniques used for CPUs and GPUs.

6.3.1.1 Register Blocking

Register blocking [56] is one of the most effective optimization techniques for SpMV on CPUs, and is central to many sparse matrix formats [21, 23, 120]. In this section, we explain why this technique is not appropriate for KNC. In register blocking, adjacent nonzeros of the matrix are grouped into small dense blocks to facilitate SIMD, and to reuse portions

of the vector x in registers. Given that KNC has wider SIMD and larger register files than previous x86 CPUs, it seems advantageous to use large blocks on KNC. However, sparse matrices in general cannot be reordered to give large dense blocks; zero padding is necessary, resulting in redundant computations with zeros and low SIMD efficiency.

To evaluate the use of register blocking, we measure the *average nonzero density of the blocks* (ratio of the number of nonzeros in a block to the number of nonzeros stored including padding zeros) using various block sizes and compare these with the minimum required SIMD efficiency (η_{min}) in Equation 31. In general, the SIMD efficiency of SpMV is approximately equal to the average nonzero density. It can be shown that, even when the smallest block sizes for KNC to facilitate SIMD are used, the nonzero densities for many matrices are still smaller than η_{min}. Thus register blocking leads to low SIMD efficiency on KNC. The details are as follows.

First, we calculate η_{min} for register blocking using Equation 31. In register blocking, all nonzeros within a block share one coordinate index, thus the working set for one nonzero is roughly equal to the number of bytes for storing one double-precision word, i. e., 8 bytes. To multiply 8 nonzeros using register blocking, an efficient implementation of the basic SIMD sequence requires approximately 6 cycles (1 cycle for loading x, 1 cycle for loading *val*, 1 cycle for multiplication, and 2-3 cycles for loop overheads and prefetching instructions). Additionally, we use 60 cores at 1.09 GHz, and the STREAM bandwidth on KNC achieves 163 GB/s. Putting these quantities together and using Equation 31, we obtain $\eta_{min} = 23.7\%$. We now measure the nonzero densities for the smallest block sizes that facilitate SIMD on KNC. The results are displayed in Table 28, showing that, for 4×4, 8×2 and 8×8 register blocks, 6, 7 and 8 matrices have nonzero density lower than 23.7%, respectively. We note that this is a crude study because no matrix reordering was used, but the above illustrates the point.

Table 28: Average percent nonzero density of register blocks and ELLPACK slices (slice height 8).

Matrix	Register blocking			SELLPACK
	4×4	8×2	8×4	
Dense8	100.00	100.00	100.00	100.00
Wind Tunnel	69.83	63.47	59.91	98.96
Circuit5M	29.70	21.82	18.39	60.07
Rail4284	20.09	14.00	10.75	38.18
Mip1	82.08	77.52	73.30	90.10
In-2004	37.99	37.59	28.19	66.69
Si41Ge41H72	30.30	25.38	20.45	64.88
Ldoor	59.60	53.64	45.98	89.75
Bone010	58.83	51.71	45.98	94.02
Rucci1	12.26	18.11	9.32	98.55
Cage15	17.25	12.47	9.98	97.16
Rajat31	12.22	10.98	6.67	98.73
12month1	9.82	8.86	5.66	25.39
Spal_004	23.12	14.00	13.66	91.39
Crankseg_2	52.17	48.72	40.33	89.76
Torso1	73.52	69.55	62.85	93.42

6.3.1.2 ELLPACK and Sliced ELLPACK

The ELLPACK format and its variants are perhaps the most effective formats for GPUs. ELLPACK packs the nonzeros of a sparse matrix towards the left and stores them in a $m \times L$ dense array (the column indices are stored in a companion array) where m is the row dimension of the matrix, and L is the maximum number of nonzeros in any row. When rows have fewer than L nonzeros, they are padded with zeros to fill out the dense array. The dense array is stored in column-major order, and thus operations on columns of the dense array can be vectorized with SIMD operations, making this format very suitable for architectures with wide SIMD.

The efficiency of the ELLPACK format highly depends on the distribution of nonzeros. When the number of nonzeros per row varies considerably, the performance degrades due to the overhead of the padding zeros. To address this problem, Monakov et al. [83] proposed the sliced ELLPACK (SELLPACK) format for GPUs, which partitions the sparse

matrix into row slices and packs each slice separately, thus requiring less zero padding than ELLPACK.

Table 28 also shows the average nonzero density of slices in SELLPACK using slice height 8, corresponding to KNC's double precision SIMD width. For matrices with highly variable numbers of nonzeros per row, the SELLPACK format still requires excessive zero padding.

6.3.2 Proposed Matrix Format

SpMV in the CSR format suffers from low SIMD efficiency, particularly for matrices with short rows. Since SIMD efficiency is expected to be higher in the ELLPACK format, and since the format supports wide SIMD operations, it forms the basis of our proposed matrix format, called *ELLPACK Sparse Block* (ESB). At a high level, ESB effectively partitions the matrix coarsely by rows and columns into large sparse blocks. The row and column partitioning is a type of cache blocking, promoting locality in accessing vector x. The sparse blocks in a block column are appended and stored in a variant of the ELLPACK format.

In practice, matrices generated by an application code may have an ordering that is "local," e. g., rows corresponding to nearby grid points of a mesh are ordered together, which translates to locality when accessing the vector x. Such an ordering may also be imposed by using a bandwidth reducing ordering. The coarse row and column partitionings in ESB are intended to maintain locality, as described below.

6.3.2.1 SELLPACK Storage with a Bit Array

The original ITPACK-ELLPACK format was designed for classical vector processors which required long vector lengths for efficiency. SIMD processing on GPUs requires vector lengths equal to the number of threads in a thread block, and thus it is natural to store the matrix in slices (i. e., SELLPACK) which requires less zero padding. SIMD processing on CPUs including KNC only requires vector lengths equal to the width of SIMD registers.

Again it is natural to store the matrix in slices, and the smaller slice height required on CPUs can make this storage scheme particularly effective at reducing zero padding.

To further reduce the memory bandwidth associated with the padding zeros, we can avoid storing zeros by storing instead the length (number of nonzeros) of each *row*, as done in ELLPACK-R [114] for GPUs. Here, each CUDA thread multiplies elements in a row until the end of the row is reached. For SIMD processing on CPUs, we can store the length of each *column* of the dense ELLPACK arrays, corresponding to the length of the vector operations for processing that column (this assumes that the rows are sorted by length and thus the nonzeros in each column are contiguous).

For KNC, instead of storing column lengths, it is more efficient to store a bit array, with ones indicating nonzero locations in the SELLPACK dense array with slice height 8. (Rows within a slice also do not need to be sorted by length when a bit array is used.) Each 8 bits corresponds to one column of a matrix slice and can be used expediently as a write-mask in KNC's SIMD instructions to dynamically reconstruct the SELLPACK dense array as the SpMV operation is being carried out. Required memory bandwidth is reduced with this bit array technique, but it technically does not increase SIMD efficiency because we still perform arithmetic on padding zeros. We note that earlier work has also used bit arrays for storing the sparsity pattern of blocks of sparse matrices in order to avoid explicitly storing zeros [121, 21].

6.3.2.2 Finite-Window Sorting

On GPUs, row sorting can be used to increase the nonzero density of slices for SELLPACK [24, 68]. Rows of the sparse matrix are sorted in descending order of number of nonzeros per row. As adjacent rows in the sorted matrix have similar numbers of nonzeros, storing the sorted matrix in SELLPACK format requires less zero padding than without sorting.

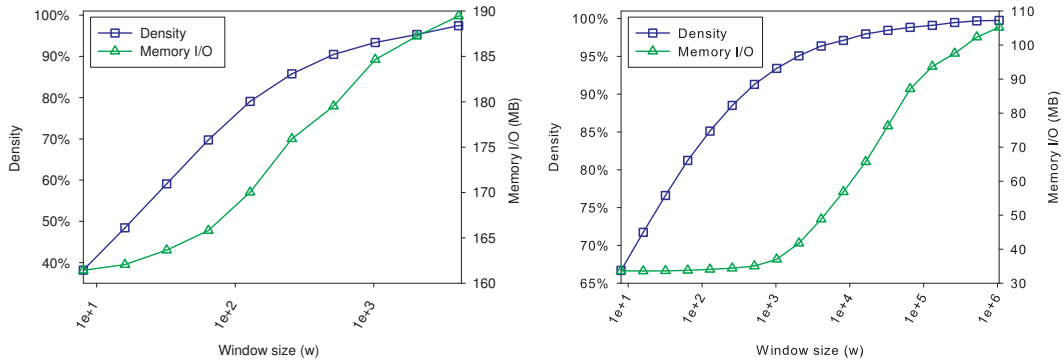
The adjacent rows in the sparse matrix in the original ordering often have high temporal

locality on accesses to vector x. This is because the matrix either naturally has good locality or has been reordered to reduce the matrix bandwidth. While row sorting increases the nonzero density, it may destroy this locality, which results in extra cache misses. To limit this side effect of row sorting, instead of sorting the entire matrix, we partition the matrix into row blocks with block height w and sort the rows within each block individually. We call this new sorting scheme *Finite-Window Sorting* (FWS). We refer to w as the *window size*. As mentioned, this is a row partitioning that promotes locality when accessing vector x.

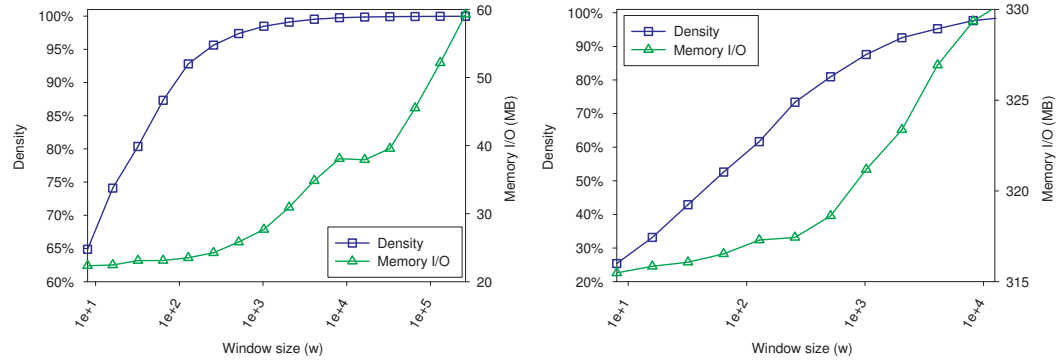
To evaluate FWS, we measure the average nonzero density of SELLPACK slices for various window sizes and the corresponding amount of memory transfer due to the accesses on vectors x and y (measured by Intel VTune) of the SpMV kernel using SELLPACK format. As an example, the results for four matrices are shown in Figure 25 We see that the nonzero density and the amount of memory transfer both increase as the window size increases. However, an important observation is that when the window size is smaller than some value, e. g., 1000 for *In-2004*, the amount of memory transfer appears to grow much more slowly than the nonzero density. This is a positive result since it suggests that we may use FWS to significantly increase the nonzero density while only slightly increasing the amount of memory transfer.

To address the low SIMD efficiency problem of SELLPACK, we use FWS to increase the average nonzero density of slices to be just larger than η_{min}. Since η_{min} is relatively small, a small w is sufficient for most matrices, thus the increase in memory transfer is very slight. We use the same w for the entire matrix.

Figure 26 illustrates the ESB format for a $m \times n$ matrix. The ESB format consists of six arrays: *val*, the nonzero values of the matrix; *colidx*, the column indices for the nonzeros; *vbit*, an array of bits marking the position of each nonzero in SELLPACK; *yorder*, the row ordering after FWS (c permutation vectors of length m); *sliceptr*, a list of $m/8 \times c$ values indexing where each slice starts in *val*; *vbitptr*, a list of c values indexing where



(a) *Rail4284* (b) *In-2004*



(c) *Si41Ge41H72* (d) *12month1*

Figure 25: Nonzero density and memory I/O as a function of window size (w).

each column block starts in *vbit*.

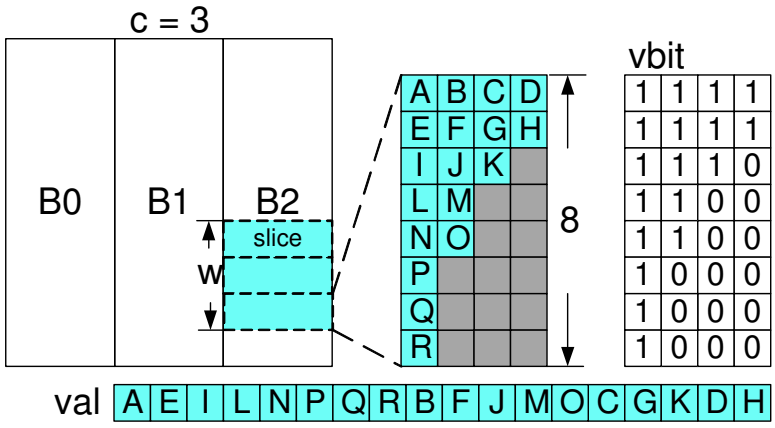


Figure 26: ELLPACK sparse block format. The sparse matrix is partitioned into c column blocks. Each column block is sorted by FWS with window size w and stored in SELLPACK format (slice height = 8) with a bit array.

6.3.3 SpMV Kernel with ESB Format

The multiplication of a column block of ESB matrix is illustrated in Algorithm 12. Due to the use of the bit array, every SIMD instruction in the inner loop, which processes one slice, has a write-mask. At the end of each inner iteration, the *popcnt* instruction (line 17) is used to count the number of 1's in *vbit*, which is equal to the number of nonzeros processed in this iteration. Since the rows of the matrix have been permuted, we use gather and scatter instructions to load and store y, using the offsets stored in $yorder$.

Algorithm 12: Multiply the i-th column block of matrix in ESB format.

```
1  yr ← i × m;
2  startslice ← i × m/8;
3  endslice ← (i+1)m/8;
4  sliceidx ← startslice;
5  idx ← sliceptr [sliceidx];
6  while sliceidx < endslice do
7      k ← vbitptr [i];
8      end ← sliceptr [sliceidx + 1];
       /* compute for a slice */
9      vec_offsets ← load (&yorder [yr]);
10     vec_y ← gather (vec_offsets, &y [0]);
11     while idx < end do
12         writemask ← vbit [k];
13         vec_idx ← loadunpack (&colidx [idx], writemask);
14         vec_vals ← loadunpack (&val [idx], writemask);
15         vec_x ← gather (vec_idx, &x [0], writemask);
16         vec_y ← fmadd (vec_vals, vec_x, vec_y, writemask);
17         idx ← idx+ popcnt (vbit [k]);
18         k ← k + 1;
19     scatter (vec_y, vec_offsets, &y [0]);
20     yr ← yr + 8;
21     sliceidx ← sliceidx + 1;
```

The reconstruction of the dense arrays are implemented using load unpack instructions (lines 13 and 14), which are able to load contiguous elements from memory and write them sparsely into SIMD registers. The write-mask in the load unpack instruction is used to mark which positions of the SIMD register to write the loaded elements.

In practice, we also parallelize the SpMV operation across column blocks. Here, we use a temporary copy of y for each block and use a reduce operation across these temporary copies at the end of the computation.

Given the ESB kernel shown in Algorithm 12, we can calculate η_{min} for the ESB format. From the ESB kernel assembly code, the basic SIMD sequence (lines 11-18) takes approximately 26 cycles. In ESB format, the working set for one nonzero includes one value and one column index, i. e., 12 bytes. Using Equation 31, we have that the SIMD efficiency of any SpMV kernel using the ESB format needs to be larger than 68.3%.

6.3.4 Selecting *c* and *w*

To select the number of block columns c and the window height w, several values are tested for each matrix. Although c and w are not independent, we find that they are only weakly related and that it is better to select c before selecting w.

The choice of number of block columns c is dependent on the structure of the sparse matrix. Nishtala et al. [85] describes situations where cache blocking is beneficial, including the presence of very long rows. We found column blocking to be beneficial for three of our test matrices: *Rail4284*, *12month1* and *Spal_004*.

For matrices that favor cache blocking, the execution time of SpMV can be modeled as $T(c) = (12nnz + 16mc + M_x(c))/B$, where the first two terms in the numerator represent the memory traffic due to accessing the matrix and vector y respectively, and $M_x(c)$ is the memory traffic due to accessing vector x. Since M_x decreases as c increases and m is relatively small, this model suggests that the execution time of the SpMV kernel might first decrease as c decreases until a certain value, near where there is no more temporal locality of vector x to exploit. After that point, the execution time will increase since the memory traffic due to accessing vector y increases as c increases.

Figure 27 shows the performance of the ESB kernel for the three matrices that favor cache blocking as c varies between 1 and 1024, which confirms the implication of our

performance model. The figure also shows that the performance decreases slowly when c is larger than the optimal value. We thus use the heuristic that we can safely choose c as a power of 2.

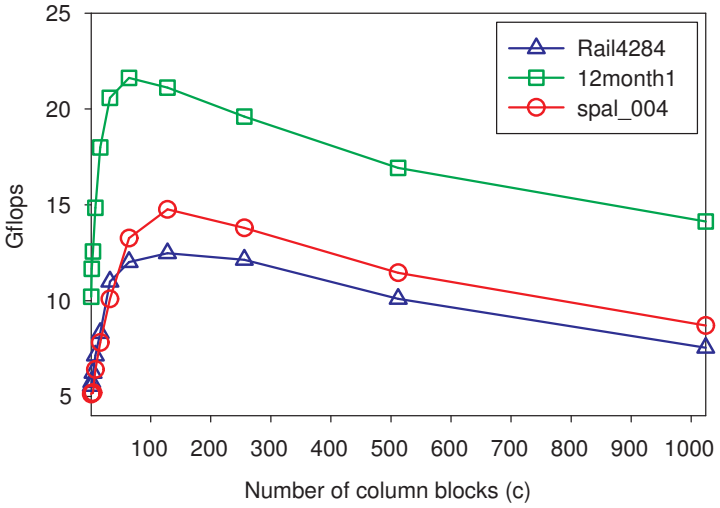


Figure 27: Performance of ESB kernel for *Rail4284 12month1*, and *Spal_004* as a function of the number of column blocks (c).

We select the window size w by gradually increasing it until the nonzero density of slices exceeds η_{min}. As discussed in Section 6.3.3, η_{min} of ESB format is 68.3%, however, we use 75% to be more conservative. The cost of search can be reduced by a matrix sampling scheme, proposed in [56] for choosing register block sizes. In the matrix sampling scheme, 1% of the slices are selected from the SELLPACK matrix to form a new matrix, and the new matrix is sorted to estimate the nonzero densities for various w.

6.4 Load Balancers for SpMV on Intel Xeon Phi

With a large number of cores, one critical performance obstacle of SpMV on KNC is load imbalance. To address this problem, we present three load balancers for SpMV and compare them with other load balancing methods. SpMV is parallelized across 240 threads on KNC, or 4 threads per core. To assign work to each thread, we consider three partitioning/load balancing schemes. The starting point for these schemes is a matrix in ESB

format, where parameters such as c, w, and matrix ordering are considered fixed. We note that these balancers are general and may be applied to other KNC workloads.

6.4.1 Static Partitioning of Cache Misses

The performance of SpMV kernels highly depends on the amount of memory transfer and the number of cache misses. This leads to our first load balancer, called *static-miss*. Here, the sparse matrix is statically partitioned into row blocks that have equal numbers of L2 cache misses. This is accomplished approximately using a simple cache simulator which models key characteristics of the L2 cache of a single core of KNC: total capacity, line size and associativity.

The main drawback of *static-miss* is the high cost of cache simulation for each matrix of interest. In our implementation, the cost is greater than that of a single matrix-vector multiplication, which means that the application of *static-miss* is limited to cases where the SpMV kernel is required to execute a large number of times with the same matrix. Like other static partitioning methods, *static-miss* also does not capture dynamic runtime factors that affect performance, e. g., the number of cache misses is sometimes highly dependent on thread execution order.

6.4.2 Hybrid Dynamic Scheduler

A more sophisticated load balancing method for SpMV on KNC is to use dynamic schedulers, which are categorized into two types: work-sharing schedulers and work-stealing schedulers. However, both types have performance issues on KNC. For work sharing schedulers, the performance bottleneck is the contention for the public task queue. Due to the large number of cores, data contention on KNC is more expensive than on multi-core CPUs, especially when data is shared by many threads. On the other hand, work stealing schedulers on KNC suffer from the high cost of stealing tasks from remote threads. The stealing cost is proportional to the distance between the thief thread and the victim thread.

To address the performance issues of both work-sharing and work-stealing schedulers,

we design a hybrid work-sharing/work-stealing scheduler, inspired by [88]. In the hybrid dynamic scheduler, the sparse matrix is first partitioned into P tasks with equal numbers of nonzeros. The tasks are then evenly distributed to N task queues, each of which is shared by a group of threads (each thread is assigned to a single queue). Within each thread group, the work-sharing scheduler is used to distribute tasks. Since each task queue is shared by a limited number of threads, the contention overhead is lowered. When a task queue is empty, the threads on the queue steal tasks from other queues. With this design, the number of stealing operations and the overhead of stealing are reduced.

The two parameters P and N need to be carefully selected to achieve high performance. P controls the maximum parallelism and also affects the temporal locality of memory accesses. With smaller tasks, the SpMV kernel will lose the temporal locality of accessing vector x. For our test matrices, P is chosen in the range of 1000-2000, based on the size of the sparse matrix. N is a tradeoff between the costs of work sharing and work stealing and is chosen to be 20 in our SpMV implementation, i. e., 12 threads (3 cores) share one task queue.

6.4.3 Adaptive Load Balancer

In many scientific applications, such as iterative linear equation solvers, the SpMV kernel is executed multiple times for the same matrix or matrix sparsity pattern. For these applications, it is possible to use performance data from earlier calls to SpMV to repartition the matrix to achieve better load balance.

This idea, which we call adaptive load balancing, was first proposed for SpMV by Lee and Eigenmann [70]. In their load balancer, the execution time of each thread is measured after each execution of SpMV. The normalized cost of each row of the sparse matrix is then approximated as the execution time of the thread to which the row is assigned, divided by the total number of rows assigned to that thread. For the next execution of SpMV, the sparse matrix is re-partitioned by evenly dividing costs of rows.

Although Lee and Eigenmann's method was originally designed for distributed memory systems, we adapted it to KNC with minor changes. To reduce the cost of re-partitioning, we used the 1D partitioning algorithm of Pinar and Aykanat [91]. Experiments show that the adaptive tuning converges in fewer than 10 executions.

6.5 *Experimental Results*

6.5.1 Load Balancing Results

We first test the load balancing techniques discussed in Section 6.4 for parallelizing SpMV for matrices in ESB format. We compare the following load balancers:

- *static-nnz*: statically partition the matrix into blocks with approximately equal numbers of nonzeros;
- *static-miss*: statically partition the matrix into blocks with approximately equal numbers of cache misses;
- *work-sharing*: working-sharing dynamic scheduler;
- *work-stealing*: working-stealing dynamic scheduler;
- *hybrid-scheduler*: hybrid dynamic scheduler;
- *adaptive-balancer*: adaptive load balancer after 5 tuning executions.

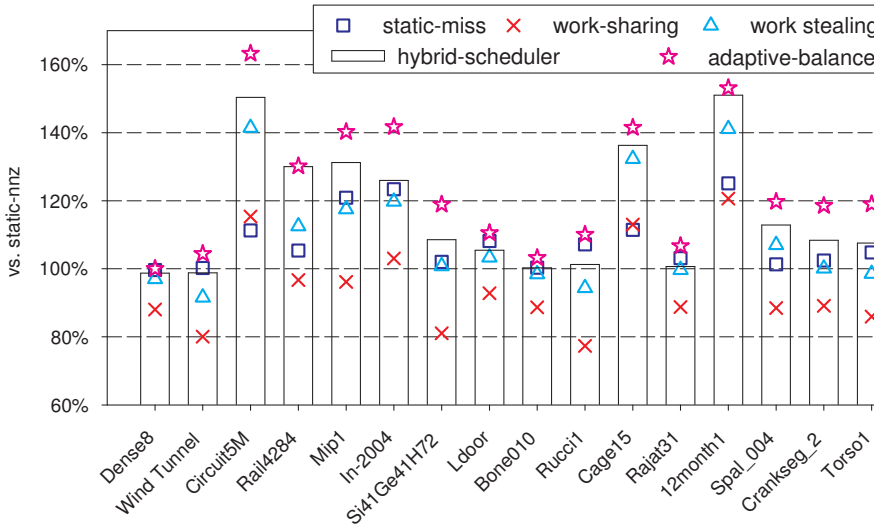


Figure 28: Normalized performance of SpMV using various load balancers.

Figure 28 shows the results, which are normalized against those of *static-nnz*. As evident in the figure, *adaptive-balancer* gives the best SpMV performance, achieving on average 1.24x improvement over *static-nnz*. The *hybrid-scheduler* has best SpMV performance among all the dynamic schedulers, but is 5% worse than *adaptive-balancer*, likely due to scheduling overheads. Results for *hybrid-scheduler* are generally better than those for *work-stealing* and *work-sharing*. The *work-sharing* scheduler is worst for most matrices and only outperforms *static-nnz* and *static-miss* on the largest matrices, e. g., *Circuit5M* and *Cage15*, for which data contention overheads can be amortized.

The static partitioning methods have an advantage over dynamic schedulers when the matrices have regular sparsity patterns, such as even distributions of the number of nonzeros per row, as in *Wind Tunnel* and *Rajat31*. We found *static-miss* to give better performance than *static-nnz* for all the matrices, but its results are not uniformly good, likely because of the poor accuracy of our simple cache simulator for irregularly structured matrices.

6.5.2 ESB Results

SpMV performance results using ESB format are presented in Figure 29. In these results, optimal values of w and c were selected using the method described in Section 6.3.4. The timings do not include time for selecting these parameters or for FWS. The hybrid dynamic scheduler was used for load balancing.

To quantify the effect of column blocking (CB), and the use of bit arrays and FWS, we also test SpMV kernels that incrementally implement these techniques. In the figure, green bars show SpMV performance using the SELLPACK format, which is our baseline. Yellow bars show the improvement by using column blocking only (applied to three matrices). Orange bars show the additional improvement by using bit arrays. Red bars show the performance using the full set of ESB optimizations.

We also use pink squares to show results using complete row sorting (no windows used for sorting). Finally, blue triangles are used to show the performance of the CSR kernel.

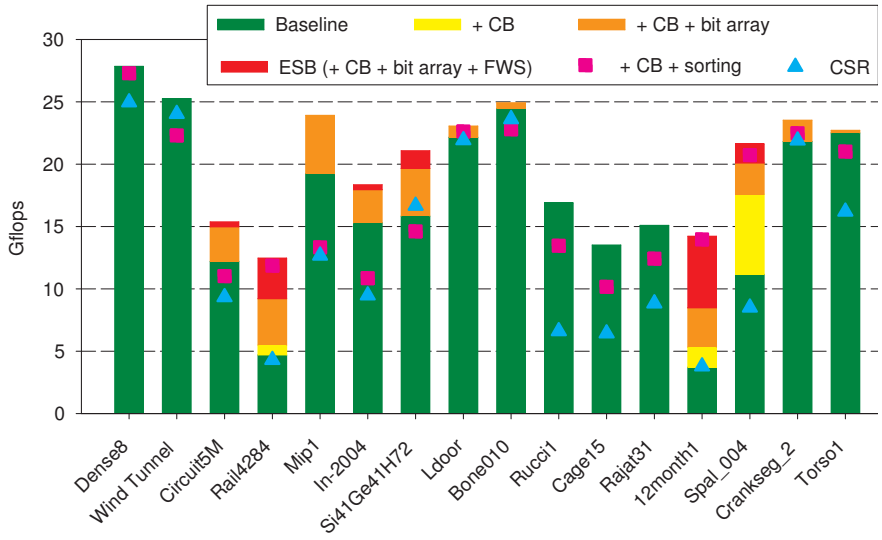


Figure 29: Performance of SpMV implementations using ESB format, showing increasing levels of optimizations of column blocking, bit array and finite-window sorting.

Although column blocking gives an improvement for the three matrices with very long rows, the improvement in two cases is very modest. A disadvantage of column blocking is that it can decrease the average nonzero density of the blocks and thus decrease SIMD efficiency.

The bit array technique improves performance in most cases, however, a small performance degradation ($< 3\%$) is also observed for some matrices, including *Dense8*, *Rucci1* and *Cage15*. This is expected for matrices with nonzero densities that are already large, since the bit array introduces additional overhead while there is no room to reduce bandwidth.

The figure also shows the significance of using FWS, which helps 6 matrices that have nonzero density lower than η_{min}. Due to its destruction of locality, complete sorting leads to poor results on KNC, with results that are poorer than the baseline for many matrices. This justifies the use of FWS.

The final ESB implementation outperforms the CSR implementation, as to be expected. On average, ESB is 1.85x faster than CSR. The greatest speedups come from the matrices with low SIMD efficiency, cache miss latency problems, and/or load imbalance.

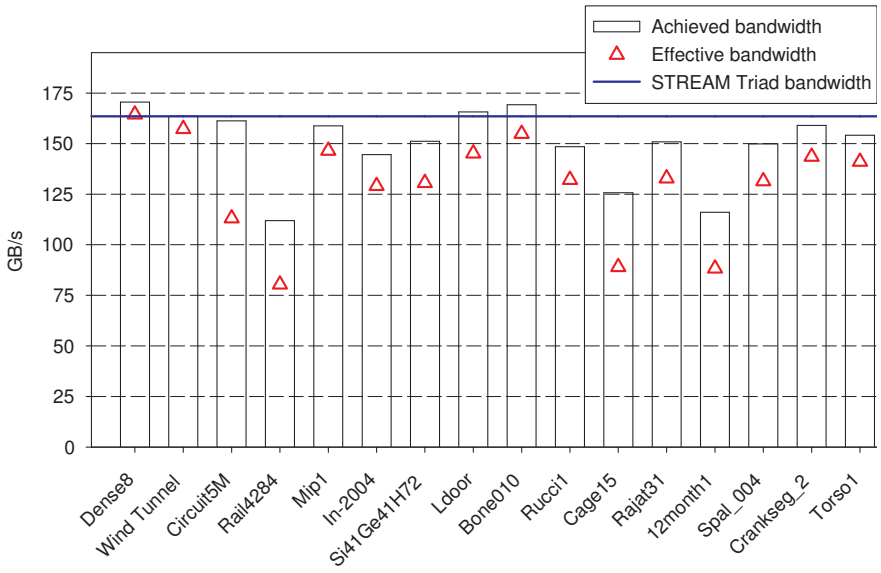


Figure 30: Achieved and effective bandwidth of SpMV implementation using ESB format.

Figure 30 shows the achieved and the effective bandwidth of our ESB kernel. The achieved bandwidth was measured by Intel VTune. The effective bandwidth, which is used in prior work [120, 24], accounts only for the "effective" memory traffic of the kernel. For a sparse matrix with m rows, n columns and nnz nonzeros, the optimistic flop:byte ratio of SpMV in ESB format is $\lambda = 2\,nnz/(12\,nnz + 16\,m + 8\,n)$, which excludes redundant memory accesses to vectors x and y. Assuming SpMV on the matrix achieves P flops, the effective bandwidth is P/λ.

By comparing the achieved and the effective bandwidth with the STREAM bandwidth, we see how far the performance of our kernel is from the upper performance bound. On average, the achieved bandwidth and effective bandwidth are within 10% and 20% of the STREAM Triad bandwidth, respectively, indicating that the implementation is very efficient.

6.5.3 Performance Comparison

We now compare the performance of SpMV on KNC with SpMV performance on the following architectures:

- *NVIDIA K20X*: NVIDIA Tesla K20X (Kepler GPU), 6 GB GDDR5 memory, 1.31 Tflops peak double precision performance, 181 GB/s (ECC on) STREAM Triad bandwidth;
- *Dual SNB-EP*: 2.7 GHz dual-socket Intel Xeon Processor E5-2680 (Sandy Brige EP), 20 MB L2 cache, 32 GB DDR memory, 346 Gflops peak double precision performance, 76 GB/s STREAM Triad bandwidth.

For the K20X platform, we used two SpMV codes: cuSPARSE v5.0[1] and CUSP v0.3 [16], which use a variety of matrix formats, CSR, COO, DIA, BCSR, ELLPACK, and ELL/HYB [15]. For each of the test matrices, we ran experiments with both codes using all the formats. For brevity, we only report the best of these results. For the dual SNB-EP system, we tested three SpMV codes: the CSR format implementation from Intel MKL v11.0, the auto-tuning implementation from Williams et al. [120] and the Compressed Sparse Block (CSB) implementation (2011 release) from Buluç et al. [21]. Again, only the best performance of the three implementations is reported. For comparison purposes, ECC was turned on in all the platforms.

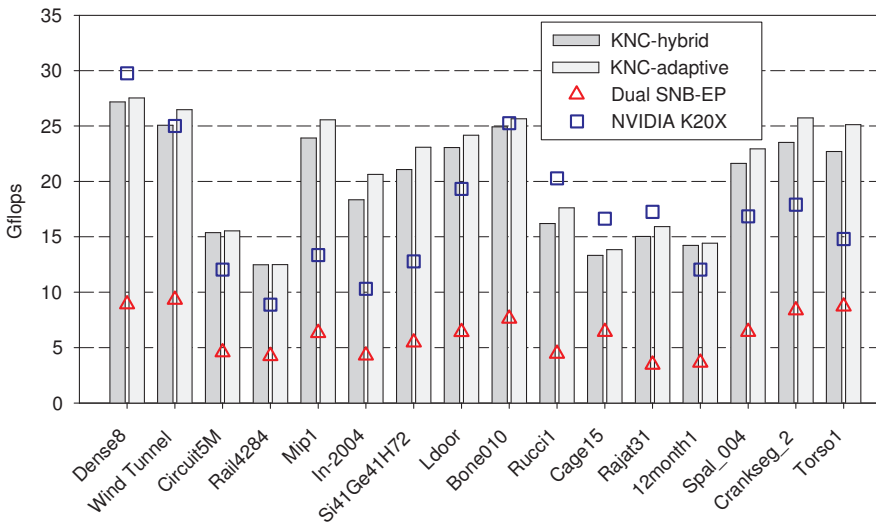


Figure 31: Performance comparison of SpMV implementations on KNC, SNB-EP and GPUs.

[1] https://developer.nvidia.com/cusparse

Figure 31 shows the comparative results. KNC has a large advantage over dual SNB-EP; for most matrices, *KNC-adaptive* and *KNC-hybrid* are 3.52x and 3.36x faster, respectively, than SNB-EP, which is expected since KNC has significantly higher memory bandwidth.

Compared to the K20X, although KNC has a lower STREAM bandwidth, *KNC-adaptive* is on average 1.32x faster than the best SpMV implementation on K20X. We believe that this mainly due to KNC having much larger caches than K20X, so accesses to vectors x and y are more likely to be directed to caches rather than memory. For the matrices with "regular" sparsity patterns, e. g., banded matrices, which are easily handled by small caches, KNC has little performance advantage over or is slower than K20X. For matrices that have complex sparsity patterns, KNC's advantage over K20X can be as high as 2x.

6.6 Summary

We have presented an efficient implementation of SpMV for the Intel Xeon Phi coprocessor. The implementation used a specialized ELLPACK-based format that promotes high SIMD efficiency and data access locality. Along with careful load balancing, the implementation achieved close to optimal bandwidth utilization. The implementation also significantly outperformed an optimized implementation using the CSR format.

There has been much recent work on SpMV for GPUs, and high performance has been attained by exploiting the high memory bandwidth of GPUs. However, GPUs are not designed for irregularly structured computations, such as operations on sparse matrices with nonuniform numbers of nonzeros per row. For our test set, we found our SpMV implementation on KNC to perform better on average than the best implementations currently available on high-end GPUs. One general explanation is that KNC has much larger caches than GPUs, which helps reduce the cost of irregular accesses on vector x. Also, additional hardware support on KNC, such as load unpack, enables more delicate optimization techniques, e. g., the bit array technique used in this work, than would be possible on GPUs

without such support. Indeed, we expect that the many compression techniques for SpMV proposed in the literature (and independent of the ideas of this work) can further reduce required bandwidth and improve the performance of our SpMV implementation, but these may be difficult to implement on GPUs.

The general performance issues raised in this work for SpMV also apply to other workloads. Due to its large SIMD width, achieving high performance on KNC will require carefully designing algorithms to fully utilize SIMD operations, even for applications that are memory bandwidth bound. Careful attention to data locality and memory access patterns can help minimize the performance impact of high cache miss latencies. Our experience with SpMV also demonstrates the importance of a well-designed load balancing method.

Chapter VII

CONCLUSIONS

In this thesis, we have presented both high performance algorithms and implementations for enabling large-scale molecular simulation applications. Our work mainly focused on two important computational problems: Hartree-Fock (HF) calculations and Brownian/Stokesian dynamics (BD/SD) simulations. First, we presented a new scalable parallel algorithm for Fock matrix construction, which is a fundamental kernel in many quantum chemistry calculations, including the HF method and Density Functional Theory (DFT). Our new algorithm for Fock matrix construction reduces communication and has better load balance than other current codes. In practice we showed that the new algorithm has nearly linear speedup and better scalability than NWChem on chemical systems that stress scalability.

We then proceeded to describe an efficient implementation of HF for large-scale distributed systems. To improve scalability and reduce time to solution, we have 1) optimized integral calculations for CPUs and Intel Xeon Phi, 2) employed a purification algorithm for computing the density matrix that scales better than diagonalization approaches, 3) developed efficient partitioning and dynamic scheduling techniques. The experiments showed that our HF implementation is able to compute 1.64 trillion electron repulsion integrals per second on 8,100 nodes of the Tianhe-2 supercomputer. Our time to solution is $10.1\times$ faster than for NWChem for 1hsg_45 on only 576 nodes of Tianhe-2. The improvement is expected to be greater for more nodes.

Based on our work on HF and Fock matrix construction, we have developed and released an open-source software, called GTFock (see Appendix B), for distributed Fock matrix computation. Because many quantum mechanical methods can be formulated in terms of (generalized) Coulomb and Exchange matrices, GTFock may form the core of

future massively parallel codes for methods including symmetry-adapted perturbation theory (SAPT), configuration interaction singles (CIS) for excited electronic states, coupled-perturbed Hartree-Fock or DFT for analytical energy gradients, and others.

In order to address the problems of large-scale hydrodynamic BD simulations, we proposed a matrix-free approach that replaces the mobility matrix by a particle-mesh Ewald (PME) summation and uses the Krylov subspace method to compute Bronwnian displacements which outperforms the conventional in terms of both the computational complexity and the memory requirement. The matrix-free approach also allows large-scale BD simulations to be accelerated on hardware that have relatively low memory capacities. In Chapter 4, we also presented an efficient implementation of the matrix-free BD algorithm on the hybrid system using the Intel Xeon Phi coprocessor. The experimental results showed that the matrix-free algorithm implemented on CPUs is more than 35x faster than the conventional BD algorithm in simulating large systems. The hybrid implementation using the matrix-free approach on two Intel Xeon Phi coprocessors achieves additional speedups of over 3.5x for large simulated systems. Software based on this work has been released in open-source form as the StokesDT (see Appendix C) package for performing large-scale Brownian and Stokesian dynamics simulations. StokesDT is capable of simulating systems with as many as 500,000 particles, while the existing BD codes are limited to approximately 3,000 particles.

In Chapter 5, we presented and tested a novel algorithm that can exploit the generalized sparse matrix-vector multiplication for many types of dynamical simulations, even though the right-hand sides are only available sequentially. In practice we measured a 30 percent speedup in performance for Stokesian dynamics simulations. The new algorithm can be regarded as an instance of a technique or approach that is applicable to other situations.

Lastly, we considered the design of efficient sparse matrix-vector multiplication (SpMV) on the Intel Xeon Phi coprocessor. SpMV is an essential kernel of many molecular simulation applications. We designed a novel sparse matrix format, called ELLPACK Sparse

Block (ESB), which is tuned for Intel Xeon Phi. Using the ESB format, our optimized SpMV implementation achieves close to 90% of the STREAM Triad bandwidth of Intel Xeon Phi and is 1.85x faster than the optimized implementation using CSR format. Compared to other architectures, SpMV on Intel Xeon Phi is 3.52x faster than on dual-socket Intel Xeon Processor E5-2680 and is 1.32x faster than on NVIDIA Tesla K20X. The ESB format has been used in the Intel MKL SpMV Format Prototype Package[1].

[1] https://software.intel.com/en-us/articles/the-intel-math-kernel-library-sparse-matrix-vector-multiply-format-prototype-package

Appendix A

OVERVIEW OF INTEL XEON PHI

A.1 Knights Corner Architecture

The Intel Xeon Phi Coprocessor, codenamed Knights Corner (KNC), is the first commercial release of the Intel Many Integrated Core (Intel MIC) architecture. Unlike previous microprocessors from Intel, KNC works on a PCIe card with GDDR5 memory and offers extremely high memory bandwidth. The first model of KNC has 60 cores, featuring a new 512-bit SIMD instruction set. With a clock speed in excess of 1 GHz, KNC has over 1 Tflops double precision peak performance from a single chip.

KNC consists of x86-based cores, caches, Tag Directories (TD), and GDDR5 Memory Controllers (MC), all connected by a high speed bidirectional ring. An architectural overview of KNC is given in Figure 32.

Each core of KNC is composed of four components: an in-order dual-issue pipeline with 4-way simultaneous multi-threading (SMT), which is modified from the *P54C* design used in the original Pentium; a 512-bit wide SIMD engine called Vector Processing Unit (VPU); 32 KB L1 data and instruction caches; and a 512 KB fully coherent L2 cache. KNC supports both the x86 instruction set and the Intel Initial Many Core Instructions (Intel IMCI), which is a new 512-bit SIMD instruction set.

The KNC's L2 caches are fully coherent using a set of tag directories. Each of the tag directories responds to L2 cache misses from a fixed portion of the memory space. Every tag directory of KNC is shared by all the cores, and the memory addresses are uniformly distributed across the tag directories. Besides the cores and tag directories, there are also GDDR memory controllers (MC) connected to the ring. As shown in Figure 32, the memory controllers are symmetrically interleaved with pairs of cores around the ring.

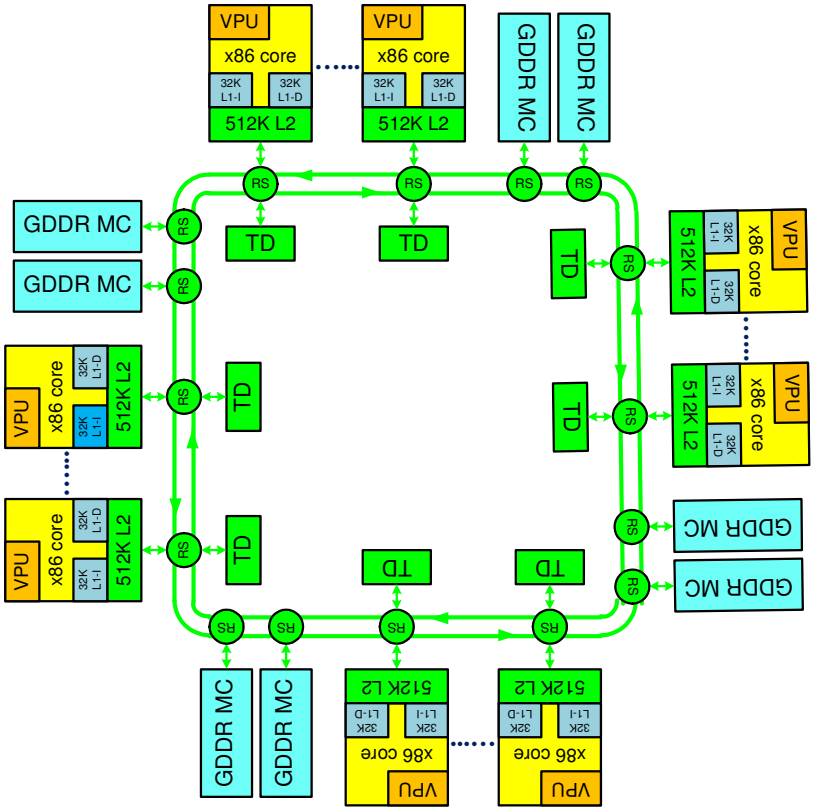


Figure 32: High-level block diagram of KNC.

The memory addresses are uniformly distributed across the tag directories and memory controllers. This design choice provides a smooth traffic characteristic on the ring and is essential for good bus performance.

Given the large number of cores and coherent caches, L2 cache misses on KNC are expensive compared to those on other x86 processors. On a L2 cache miss, an address request is sent on the ring to the corresponding tag directory. Depending on whether or not the requested address is found in another core's cache, a forwarding request is then sent to that core or memory controllers, and the request data is subsequently forwarded on the ring. The cost of each data transfer on the ring is proportional to the distance between the source and the destination, which is, in the worse case, on the order of hundreds of cycles. Overall, the L2 cache miss latency on KNC can be an order of magnitude larger than that of multi-core CPUs.

A.2 Intel Initial Many Core Instructions

KNC provides a completely new 512-bit SIMD instruction set called Intel Initial Many Core Instructions (Intel IMCI). Compared to prior vector architectures (MMX, SSE, and AVX), Intel IMCI has larger SIMD width and introduces many new features. One important new feature is operand *modifier*, which is able to modify the operands on the fly when the instruction is being executed. Most IMCI instructions support two types of modifiers. The optional *swizzle* modifier causes elements of the second source vector in a SIMD instruction to be permuted when the instruction is being executed. The other modifier is *write-mask*. The write-mask in a SIMD instruction is a mask register coming with the destination vector. After the SIMD instruction is executed, each element in the destination vector is conditionally updated with the results of the instruction, contingent on the corresponding element position bit in the mask register.

Another new feature offered by Intel IMCI is *load unpack* and *store pack* instructions. The premise behind them is that, for many applications, memory is accessed contiguously, but contents read or written with respect to the vector register are not contiguous. In order to achieve this, the write-mask registers in load unpack and store pack instructions are used in a slightly different manner. As an example, the behavior of the load unpack instruction is shown in Figure 33. Load unpack and store pack instructions can be also used to load and store unaligned elements since the memory addresses of them are not necessary to be aligned to 512-bit boundary.

Additionally, Intel IMCI introduces vector *gather* and *scatter* operations to complete load and store support, which allow manipulation of irregular data patterns of memory. Specifically, the gather instruction loads sparse locations of memory into a dense vector register, while the scatter instruction performs inversely. In gather and scatter instructions, the loading or storing locations of memory are specified by a base address and a vector of signed 32-bit integers which indicate the offsets to the base address.

While the gather and scatter instructions provide great flexibility of memory reference,

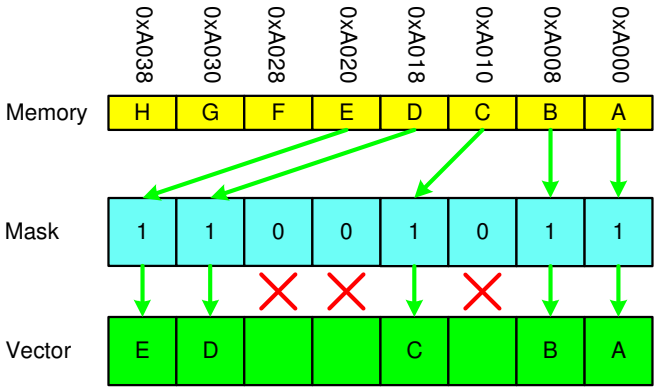


Figure 33: The behavior of the load unpack instruction.

they are very expensive. In Intel IMCI, the gather (or scatter) instruction behaves as a gather (or scatter) step operation. That is, instead of loading (or storing) all the elements from memory by issuing one instruction, the operation is defined to load (or store) at least one element from memory each time it is issued. The write-mask registers in the gather and scatter instructions are used by the hardware to track which elements are completed and which are still pending. As each element is loaded from (or stored to) memory, the corresponding element bit in the write-mask register is cleared. Thus, to gather or scatter 8 double-precision floating points, it is necessary to either issue 8 gather or scatter operations or to place the instruction inside a loop. As an example, the loop body for the gather instruction is shown in Listing A.1.

Listing A.1: Loop body to create a full gather sequence.

```
repeat:
  vgatherdpd base, zmm1, zmm2{k2} ; gather 8
      values from base to zmm2 and the offsets
      are indicated by zmm1. k2 is the
      write-mask.
  jknzd repeat, k2                ; any
      elements not gathered yet? If so, repeat
```

Appendix B

GTFOCK DISTRIBUTED FRAMEWORK

GTFock is a distributed parallel framework for Fock matrix construction, which 1) uses a new scalable parallel algorithm that significantly reduces communication and has better load balance than other current nodes, 2) provides a generalized computational interface for constructing Fock matrices which can be easily used to build customized Hartree-Fock or Density Functional Theory applications. and 3) can compute multiple Fock matrices on one run and handle non-symmetric density matrices. GTFock can be downloaded at https://code.google.com/p/gtfock/.

B.1 Installation

In this section, a step-by-step description of the build process and necessary and optional environment variables is outlined.

1. Setting up the proper environment variables

   - *WORK_TOP* defines the work directory.

     ```
     export WORK_TOP=$PWD
     ```

   - *GA_TOP* defines where the Global Arrays library is (going to be) installed.

     ```
     export GA_TOP=$WORK_TOP/GAlib
     ```

   - *ARMCI_TOP* defines where the ARMCI library is (going to be) installed.

     ```
     export ARMCI_TOP=$WORK_TOP/external-armci
     ```

   - *ERD_TOP* defines where the OptERD library is (going to be) installed.

     ```
     export ERD_TOP=$WORK_TOP/ERDlib
     ```

2. Installing Global Arrays on MPI-3

   - Installing armci-mpi

```
cd $WORK_TOP
git clone git://git.mpich.org/armci-mpi.git || \
    git clone http://git.mpich.org/armci-mpi.git
cd armci-mpi
git checkout mpi3rma
./autogen.sh
mkdir build
cd build
../configure CC=mpiicc --prefix=$WORK_TOP/external-armci
make -j12
make install
```

   - Installing Global Arrays

```
cd $WORK_TOP
wget http://hpc.pnl.gov/globalarrays/download/ga-5-3.tgz
tar xzf ga-5-3.tgz
cd ga-5-3
# carefully set the mpi executables that you want to use
./configure CC=mpiicc MPICC=mpiicc CXX=mpiicpc \
    MPICXX=mpiicpc F77=mpiifort MPIF77=mpiifort \
    FC=mpiifort MPIFC=mpiifort --with-mpi \
    --with-armci=$WORK_TOP/external-armci \
    --prefix=$WORK_TOP/GAlib make -j12 install
```

3. or Installing Global Arrays on ARMCI

```
cp config_ga.py $WORK_TOP
# openib means using Infiniband
```

```
python config_ga.py download openib
```

4. Installing the OptErd library in *$WORK_TOP/ERDlib*

```
cd $WORK_TOP
svn co https://github.com/Maratyszcza/OptErd/trunk OptErd
cd $WORK_TOP/OptErd
make -j12
prefix=$WORK_TOP/ERDlib make install
```

5. Installing GTFock in *$WORK_TOP/gtfock/install/*

```
cd $WORK_TOP
svn co http://gtfock.googlecode.com/svn/trunk gtfock
cd $WORK_TOP/gtfock/

# Change the following variables in make.in
BLAS_INCDIR      = /opt/intel/mkl/include/
BLAS_LIBDIR      = /opt/intel/mkl/lib/intel64/
BLAS_LIBS        = -lmkl_intel_lp64 -lmkl_core \
                   -lmkl_intel_thread -lpthread -lm
SCALAPACK_INCDIR = /opt/intel/mkl/include/
SCALAPACK_LIBDIR = /opt/intel/mkl/lib/intel64/
SCALAPACK_LIBS   = -lmkl_scalapack_lp64 \
                   -lmkl_blacs_intelmpi_lp64
MPI_LIBDIR = /opt/mpich2/lib/
MPI_LIBS =
MPI_INCDIR = /opt/mpich2/include

make
```

B.2 API Reference

GTFock provides a flexible interface for constructing Fock matrices on distributed systems. A typical invocation of GTFock is composed of the following five steps.

1. Creating a PFock computing engine

```
/**
 * Create a PFock computing engine on 3x3 MPI processes.
 * Each MPI process owns 16 (4x4) tasks.
 * The screening threshold is 1e-10. The maximum number
 * of density matrices can be computed is 5. The input
 * density
 * matrices can be non-symmetric.
 **/
PFock_t pfock;
PFock_create(basis, 3, 3, 4, 1e-10, 5, 0, &pfock);
```

2. Setting the number of density matrices to be computed

```
// The number of Fock matrices to be computed is 3.
PFock_setNumDenMat(3, pfock);
```

3. Putting local data into the global densities matrices

```
/**
  * Put the local data onto the range
  * (rowstart:rowend, colstart:colend)
  * of the global density matrix 2.
 **/
PFock_putDenMat(rowstart, rowend, colstart,
                colend, ld, localmat, 2, pfock);
PFock_commitDenMats(pfock);
```

4. Computing Fock matrices

```
PFock_computeFock(basis, pfock);
```

5. Getting data from the global Fock matrices

```
/**
  * Get the local data from the range
  * (rowstart:rowend, colstart:colend)
  * of the global Fock matrix 1.
 **/
PFock_getMat(pfock, PFOCK_MAT_TYPE_F, 1, rowstart,
             rowend, colstart, colend, stride, F_block);
```

B.3 Example SCF code

GTFock also includes an example SCF code, which demonstrates how to use GTFock to build a non-trivial quantum chemistry application. The example SCF code uses a variant of McWeeny purification, called canonical purification, to compute the density matrix, which scales better than diagonalization approaches. In the example SCF code, a 3D matrix-multiply kernel is implemented for efficient purification calculations.

The example SCF code can be run as

```
mpirun -np <nprocs> ./scf <basis> <xyz> <nprow> <npcol> \
    <np2> <ntasks> <niters>
```

- nprocs: the number of MPI processes
- basis: the basis file
- xyz: the xyz file
- nprow: the number of MPI processes per row

- npcol: the number of MPI processes per col
- np2: the number of MPI processes per one dimension for purification (eigenvalue solve)
- ntasks: the each MPI process has ntasks x ntasks tasks
- niters: the number of SCF iterations

NOTE:

1. $nprow \times npcol$ must be equal to nprocs
2. $np2 \times np2 \times np2$ must be smaller than nprocs
3. suggested values for ntasks: 3, 4, 5

For example, the following command run the SCF code on graphene_12_54_114.xyz with 12 MPI processes for 10 iterations. The number of processes used for purification is $2 \times 2 \times 2 = 8$.

```
mpirun -np 12 ./pscf/scf data/guess/cc-pvdz.gbs \
    data/graphene/graphene_12_54_114.xyz 3 4 2 4 10
```

Appendix C

STOKESDT TOOLKIT

StokesDT runs on x86 based Linux systems with Pentium 4 or more recent processors. To build the source code, StokesDT requires Intel C++ compiler version 13.0 or greater and MKL version 11.0 or greater. It also requires installing the latest Maltab and the Mex compiler if the user wants to use the Matlab interface.

C.1 Installation

It is very straightforward to compile and install StokesDT. First, setup the proper environment variables in *makevars.in* in the top directory of the StokesDT source tree. For example:

```
CC     = /opt/intel/bin/icc
CXX    = /opt/intel/bin/icpc
LD     = /opt/intel/bin/xild
AR     = /opt/intel/bin/xiar
RANLIB = /opt/intel/bin/ranlib
MEX    = /opt/MATLAB/bin/mex
MEXEXT = mexa64
CXXFLAGS  = -O3 -Wall -w2
CXXFLAGS += -openmp
CXXFLAGS += -mkl
CXXFLAGS += -std=c++11
CXXFLAGS += -Wsign-compare -Wunknown-pragmas -Wreturn-type
CXXFLAGS += -Wunused-variable -Wuninitialized
CXXFLAGS += -Wmissing-prototypes -Wmissing-declarations
```

```
CXXFLAGS += -qno-offload
CXXFLAGS += -DENABLE_PROFILE_
MKLROOT = /opt/intel/mkl/
MEXFLAGS = CXXFLAGS="-O3 -Wall -fPIC -w2 -openmp \
           -std=c++11 -qno-offload\
           -I${TOPDIR}/install/include" \
           LDFLAGS="-static-intel -mkl -liomp5 -lm"
CP    = cp -f
RM    = rm -f
MKDIR = mkdir -p
```

To find the proper value of *MEXEXT*, run the function *mexext()* in Matlab.

Then, you can run the following the commands to build StokesDT.

- To compile the StokesDT libraries and the main executable,

  ```
  make
  ```

- To build and run the test programs,

  ```
  make test
  ```

- To build the Matlab interface,

  ```
  make matlab
  ```

- To build the auxiliary tools,

  ```
  make tool
  ```

All the libraries, header files and binaries will be installed in the install directory. The directory structure is shown in Table 29.

Table 29: StokesDT directory structure.

Directory	Description
install/include	header files
install/lib	StokesDT libraries
install/bin	main executable
install/tools	auxiliary tools
install/mex	Mex binaries

C.2 Running StokesDT

To run StokesDT simply use the command *stokesdt* and provide the name of the input files. StokesDT uses three input control files to specify a simulation. A detailed description of the control files can be found in Section C.3.

Generally, start a simulation with the following command:

```
./install/bin/stokesdt --config <config file> --model <model file>\
    --xyz <XYZ file> -n <number of steps to simulate>
```

For more information:

```
./install/bin/stokesdt --help
```

C.3 Control File Interface

Three input files are needed to perform simulations with StokesDT. The config file (*.cfg*) specifies all the simulation parameters, including the time step length, the type of the simulation algorithm, the types of methods for building mobility matrices and computing Brownian forces, etc. The model file (*.mod*) defines the molecular system to be simulated, including the properties of the particles, and the forces between particles. The XYZ file (*.xyz*) contains the initial positions of the particles.

The config file also specifies the optional output from a simulation, which consists of two types of files. The log file (.log) is a log of the simulation, with performance statistics, error messages and debugging information. The trajectory file, in XYZ format (.xyz),

contains the positions of the particles in the simulation at user-defined intervals. For more details, see https://code.google.com/p/stokesdt/wiki/ControlFile.

C.4 API Reference

StokesDT provides an object-oriented interface which can be easily used to develop customized hydrodynamic simulation codes. These APIs are wrapped in static libraries, which can be found in install/lib. For each main computational kernel in Brownian/Stokesian simulations, StokesDT defines a group of C++ classes, and each of them implements a different method for computing the kernel. Each group of classes generally consists of a base class that contains the abstract interface and a number of derived classes that implements the computational kernels. The base classes defined in StokesDT and their brief descriptions are shown in Table 30.

Table 30: StokesDT base classes.

Base class	Description
MobBase	Abstract base class for constructing mobility matrix
BrwnBase	Abstract base class for computing brownian displacement vectors
ForceBase	Abstract base class for computing forces
RndStream	Random number generator
PairListBase	Abstract base class for pair list

For more details, see https://code.google.com/p/stokesdt/wiki/API

C.5 Matlab Interface

StokesDT also provides a Mex interface which allows users to use the highly optimized computational kernels of StokesDT with Matlab. A detailed description of the Matlab interface can be found at https://code.google.com/p/stokesdt/wiki/MexInterface.

REFERENCES

[1] "NWChem Web page," Retrieved May. 1, 2014. http://www.nwchem-sw.org/index.php/Benchmarks.

[2] AGARWAL, R., BALLE, S., GUSTAVSON, F., JOSHI, M., and PALKAR, P., "A three-dimensional approach to parallel matrix multiplication," *IBM J. Res. Dev.*, vol. 39, pp. 575–582, 1995.

[3] ALDER, B. J. and WAINWRIGHT, T., "Studies in molecular dynamics. i. general method," *The Journal of Chemical Physics*, vol. 31, no. 2, pp. 459–466, 1959.

[4] ALLEN, M. P. and TILDESLEY, D. J., *Computer Simulation of Liquids*. Oxford: Clarendon Press, 1989.

[5] ANDO, T., CHOW, E., and SKOLNICK, J., "Dynamic simulation of concentrated macromolecular solutions with screened long-range hydrodynamic interactions: Algorithm and limitations," *The Journal of Chemical Physics*, vol. 139, p. 121922, 2013.

[6] ANDO, T., CHOW, E., SAAD, Y., and SKOLNICK, J., "Krylov subspace methods for computing hydrodynamic interactions in Brownian dynamics simulations," *The Journal of Chemical Physics*, vol. 137, p. 064106, 2012.

[7] ANDO, T. and SKOLNICK, J., "Crowding and hydrodynamic interactions likely dominate in vivo macromolecular motion," *Proc Natl Acad Sci USA*, vol. 107, pp. 18457–18462, Oct. 2010.

[8] APRÀ, E., RENDELL, A. P., HARRISON, R. J., TIPPARAJU, V., DEJONG, W. A., and XANTHEAS, S. S., "Liquid water: obtaining the right answer for the right reasons," in *Proc. Conference on High Performance Computing Networking, Storage and Analysis*, p. 66, ACM, 2009.

[9] ASADCHEV, A., ALLADA, V., FELDER, J., BODE, B. M., GORDON, M. S., and WINDUS, T. L., "Uncontracted Rys quadrature implementation of up to g functions on graphical processing units," *Journal of Chemical Theory and Computation*, vol. 6, no. 3, pp. 696–704, 2010.

[10] ASHWORTH, M., BUSH, I. J., D'MELLOW, K., HEIN, J., HENTY, D., GUEST, M. F., PLUMMER, M., SMITH, L., SUNDERLAND, A. G., SIMPSON, A., and TREW, A., "Capability incentive analysis of phase 2A codes," Tech. Rep. HPCxTR0612, University of Edinburgh HPCx Ltd, Edinburgh EH9 3JZ, 2006.

[11] BALL, R. C. and MELROSE, J. R., "A simulation technique for many spheres in quasistatic motion under frame-invariant pair drag and Brownian forces," *Physica A*, vol. 247, p. 444, 1997.

[12] BARTLETT, R. J., "Many-body perturbation theory and coupled cluster theory for electron correlation in molecules," *Annual Review of Physical Chemistry*, vol. 32, no. 1, pp. 359–401, 1981.

[13] BAUMGARTNER, G., AUER, A., BERNHOLDT, D. E., BIBIREATA, A., CHOPPELLA, V., COCIORVA, D., GAO, X., HARRISON, R. J., HIRATA, S., KRISHNAMOORTHY, S., and OTHERS, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proc. IEEE*, vol. 93, no. 2, pp. 276–292, 2005.

[14] BEENAKKER, C. W. J., "Ewald sums of the Rotne-Prager tensor," *The Journal of Chemical Physics*, vol. 85, pp. 1581–1582, 1986.

[15] BELL, N. and GARLAND, M., "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. ACM/IEEE Conf. Supercomputing*, SC '09, p. 18, ACM, 2009.

[16] BELL, N. and GARLAND, M., "CUSP: Generic parallel algorithms for sparse matrix and graph computations," 2012. V0.3.0.

[17] BLUMOFE, R. D. and LEISERSON, C. E., "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[18] BOSSIS, G. and BRADY, J. F., "Dynamic simulation of sheared suspensions. I. General method," *The Journal of Chemical Physics*, vol. 80, no. 10, pp. 5141–5154, 1984.

[19] BRADY, J. F. and BOSSIS, G., "Stokesian Dynamics," *Annual Review of Fluid Mechanics*, vol. 20, no. 1, pp. 111–157, 1988.

[20] BULUC, A. and GILBERT, J., "Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication," in *Proc. ICPP*, 2008.

[21] BULUC, A., WILLIAMS, S., OLIKER, L., and DEMMEL, J., "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *Proc. 2011 IEEE Intl Parallel & Distributed Processing Symposium*, IPDPS 2011, (Washington, DC, USA), pp. 721–733, IEEE, 2011.

[22] BUSH, I. J., S., T., SEARLE, B. G., MALLIA, G., BAILEY, C. L., MONTANARI, B., BERNASCONI, L., CARR, J. M., and HARRISON, N. M., "Parallel implementation of the ab initio CRYSTAL program: electronic structure calculations for periodic systems," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science*, vol. 467, no. 2131, pp. 2112–2126, 2011.

[23] BUTTARI, A., EIJKHOUT, V., LANGOU, J., and FILIPPONE, S., "Performance optimization and modeling of blocked sparse kernels," *Intl J. High Perf. Comput. Appl.*, vol. 21, pp. 467–484, 2007.

[24] CHOI, J., SINGH, A., and VUDUC, R., "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *ACM SIGPLAN Notices*, vol. 45, pp. 115–126, ACM, 2010.

[25] CICHOCKI, B., JEZEWSKA, M. L. E., and WAJNRYB, E., "Lubrication corrections for three-particle contribution to short-time self-diffusion coefficients in colloidal dispersions," *The Journal of Chemical Physics*, vol. 111, no. 7, pp. 3265–3273, 1999.

[26] DARDEN, T., YORK, D., and PEDERSEN, L., "Particle mesh Ewald – an $N\log(N)$ method for Ewald sums in large systems," *Journal of Chemical Physics*, vol. 98, pp. 10089–10092, 1993.

[27] DAVIS, J. and CHUNG, E., "SpMV: A memory-bound application on the GPU stuck between a rock and a hard place," *Microsoft Technical Report*, 2012.

[28] DEKEL, E., NASSIMI, D., and SAHNI, S., "Parallel matrix and graph algorithms," *SIAM Journal on Computing*, vol. 10, no. 4, pp. 657–675, 1981.

[29] DICKINSON, E., "Brownian dynamic with hydrodynamic interactions: the application to protein diffusional problems," *Chem. Soc. Rev.*, vol. 14, no. 4, pp. 421–455, 1985.

[30] DINAN, J., LARKINS, D. B., SADAYAPPAN, P., KRISHNAMOORTHY, S., and NIEPLOCHA, J., "Scalable work stealing," in *Proc. Conference on High Performance Computing Networking, Storage and Analysis*, p. 53, ACM, 2009.

[31] DŁUGOSZ, M., ANTOSIEWICZ, J. M., and TRYLSKA, J., "Association of aminoglycosidic antibiotics with the ribosomal A-site studied with Brownian dynamics," *Journal of Chemical Theory and Computation*, vol. 4, pp. 549–559, 2008.

[32] DŁUGOSZ, M. and TRYLSKA, J., "Diffusion in crowded biological environments: applications of Brownian dynamics," *BMC Biophysics*, vol. 4, no. 1, p. 3, 2011.

[33] DŁUGOSZ, M., ZIELIŃSKI, P., and TRYLSKA, J., "Brownian dynamics simulations on CPU and GPU with BD_BOX," *J. Comput. Chem.*, vol. 32, pp. 2734–2744, 2011.

[34] DUNNING JR, T. H., "Gaussian basis sets for use in correlated molecular calculations. I. The atoms boron through neon and hydrogen," *The Journal of Chemical Physics*, vol. 90, p. 1007, 1989.

[35] DUPUIS, M., RYS, J., and KING, H. F., "Evaluation of molecular integrals over Gaussian basis functions," *The Journal of Chemical Physics*, vol. 65, no. 1, pp. 111–116, 1976.

[36] DURLOFSKY, L., BRADY, J. F., and BOSSIS, G., "Dynamic simulation of hydrodynamically interacting particles," *Journal of Fluid Mechanics*, vol. 180, pp. 21–49, 1987.

[37] ELLIS, R. J., "Macromolecular crowding: obvious but underappreciated," *Trends in Biochemical Sciences*, vol. 26, no. 10, pp. 597–604, 2001.

[38] ERMAK, D. L. and MCCAMMON, J. A., "Brownian dynamics with hydrodynamic interactions," *The Journal of Chemical Physics*, vol. 69, pp. 1352–1360, 1978.

[39] ESSMANN, U., PERERA, L., BERKOWITZ, M. L., DARDEN, T., LEE, H., and PEDERSEN, L. G., "A smooth particle mesh Ewald method," *Journal of Chemical Physics*, vol. 103, pp. 8577–8593, 1995.

[40] FERMI, E., PASTA, J., and ULAM, S., "Studies of nonlinear problems," 1955.

[41] FIXMAN, M., "Simulation of polymer dynamics. I. General theory," *The Journal of Chemical Physics*, vol. 69, no. 4, pp. 1527–1537, 1978.

[42] FIXMAN, M., "Construction of Langevin forces in the simulation of hydrodynamic interaction," *Macromolecules*, vol. 19, pp. 1204–1207, 1986.

[43] FLOCKE, N. and LOTRICH, V., "Efficient electronic integrals and their generalized derivatives for object oriented implementations of electronic structure calculations," *Journal of Computational Chemistry*, vol. 29, no. 16, pp. 2722–2736, 2008.

[44] FOSTER, I. T., TILSON, J. L., WAGNER, A. F., SHEPARD, R. L., HARRISON, R. J., KENDALL, R. A., and LITTLEFIELD, R. J., "Toward high-performance computational chemistry: I. Scalable Fock matrix construction algorithms," *Journal of Computational Chemistry*, vol. 17, no. 1, pp. 109–123, 1996.

[45] FREMBGEN-KESNER, T. and ELCOCK, A. H., "Striking effects of hydrodynamic interactions on the simulated diffusion and folding of proteins," *Journal of Chemical Theory and Computation*, vol. 5, pp. 242–256, 2009.

[46] FRENKEL, D. and SMIT, B., *Understanding molecular simulation: from algorithms to applications*, vol. 1. Academic press, 2001.

[47] GEYER, T., "Many-particle Brownian and Langevin dynamics simulations with the Brownmove package," *BMC Biophysics*, vol. 4, p. 7, 2011.

[48] GEYER, T. and WINTER, U., "An $O(N^2)$ approximation for hydrodynamic interactions in Brownian dynamics simulations," *The Journal of Chemical Physics*, vol. 130, p. 114905, 2009.

[49] GOUMAS, G., KOURTIS, K., ANASTOPOULOS, N., KARAKASIS, V., and KOZIRIS, N., "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *J. Supercomput.*, vol. 50, pp. 36–77, 2009.

[50] GRASSIA, P. S., HINCH, E. J., and NITSCHE, L. C., "Computer simulations of Brownian motion of complex systems," *Journal of Fluid Mechanics*, vol. 282, pp. 373–403, 1995.

[51] GROPP, W., KAUSHIK, D., KEYES, D., and SMITH, B., "Toward realistic performance bounds for implicit CFD codes," in *Proceedings of Parallel CFD'99* (ECER, A., ed.), Elsevier, 1999.

[52] GUCKEL, E. K., *Large scale simulation of particulate systems using the PME method*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[53] HARRISON, R. J., GUEST, M. F., KENDALL, R. A., BERNHOLDT, D. E., WONG, A. T., STAVE, M., ANCHELL, J. L., HESS, A. C., LITTLEFIELD, R., FANN, G. I., and OTHERS, "High performance computational chemistry: II. a scalable SCF program," *J. Comp. Chem*, vol. 17, p. 124, 1993.

[54] HARVEY, M. and DE FABRITIIS, G., "An implementation of the smooth particle mesh Ewald method on GPU hardware," *Journal of Chemical Theory and Computation*, vol. 5, pp. 2371–2377, 2009.

[55] HOCKNEY, R. W. and EASTWOOD, J. W., *Computer Simulation Using Particles*. Bristol: Adam Hilger, 1988.

[56] IM, E., YELICK, K., and VUDUC, R., "SPARSITY: Optimization framework for sparse matrix kernels," *Intl J. High Perf. Comput. Appl.*, vol. 18, pp. 135–158, 2004.

[57] IM, E.-J., *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, University of California, Berkeley, Jun 2000.

[58] "Intel Advanced Vector Extensions Programming Reference." 2008, http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel-AVX-Programming-Reference-31943302.pdf.

[59] "Intel SSE4 programming reference." 2007, http://www.intel.com/design/processor/manuals/253

[60] ISHIKAWA, T., SEKIYA, G., IMAI, Y., and YAMAGUCHI, T., "Hydrodynamic interactions between two swimming bacteria," *Biophysical Journal*, vol. 93, pp. 2217–2225, 2007.

[61] JANSSEN, C. L. and NIELSEN, I. M., *Parallel computing in quantum chemistry*. CRC Press, 2008.

[62] JEFFREY, D. J. and ONISHI, Y., "Calculation of the resistance and mobility functions for two unequal rigid spheres in low-Reynolds-number flow," *Journal of Fluid Mechanics*, vol. 139, pp. 261–290, 1984.

[63] KARAKASIS, V., GOUMAS, G., and KOZIRIS, N., "A comparative study of blocking storage methods for sparse matrices on multicore architectures," in *Proc. 2009 Intl Conf. Comput. Sci. and Eng.*, CSE '09, pp. 247–256, IEEE, 2009.

[64] KARYPIS, G. and KUMAR, V., "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, 1999.

[65] KIM, S. and KARRILA, S. J., *Microhydrodynamics: Principles and Selected Applications*. Boston: Butterworth-Henemann, June 1991.

[66] KOTAR, J., LEONI, M., BASSETTI, B., LAGOMARSINO, M. C., and CICUTA, P., "Hydrodynamic synchronization of colloidal oscillators," *Proc Natl Acad Sci USA*, vol. 107, pp. 7669–7673, 2010.

[67] KOURTIS, K., GOUMAS, G., and KOZIRIS, N., "Exploiting compression opportunities to improve SpMxV performance on shared memory systems," *ACM Trans. Architecture and Code Optimization*, vol. 7, p. 16, 2010.

[68] KREUTZER, M., HAGER, G., WELLEIN, G., FEHSKE, H., BASERMANN, A., and BISHOP, A. R., "Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation," in *Proc. 2012 IEEE Intl Parallel and Distributed Processing Symposium Workshops*, IPDPS 2012, (Washington, DC, USA), pp. 1696–1702, IEEE, 2012.

[69] LEE, B. C., VUDUC, R. W., DEMMEL, J. W., and YELICK, K. A., "Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply," in *Proceedings of the 2004 International Conference on Parallel Processing*, ICPP '04, (Washington, DC, USA), pp. 169–176, IEEE Computer Society, 2004.

[70] LEE, S. and EIGENMANN, R., "Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems," in *Proc. 22nd Intl Conf. Supercomputing*, ICS '08, pp. 195–204, ACM, 2008.

[71] LIU, X., CHOW, E., VAIDYANATHAN, K., and SMELYANSKIY, M., "Improving the performance of dynamical simulations via multiple right-hand sides," in *Proc. 2012 IEEE Parallel & Distributed Processing Symposium*, pp. 36–47, IEEE, 2012.

[72] LIU, X., SMELYANSKIY, M., CHOW, E., and DUBEY, P., "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proc. 27th International Conference on Supercomputing*, pp. 273–282, ACM, 2013.

[73] LOTRICH, V., FLOCKE, N., PONTON, M., YAU, A., PERERA, A., DEUMENS, E., and BARTLETT, R., "Parallel implementation of electronic structure energy, gradient, and hessian calculations," *The Journal of Chemical Physics*, vol. 128, p. 194104, 2008.

[74] LUBY-PHELPS, K., "Cytoarchitecture and physical properties of cytoplasm: Volume, viscosity, diffusion, intracellular surface area," *International Review of Cytology – a Survey of Cell Biology*, vol. 192, pp. 189–221, 2000.

[75] LUEHR, N., UFIMTSEV, I. S., and MARTÍNEZ, T. J., "Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs)," *Journal of Chemical Theory and Computation*, vol. 7, no. 4, pp. 949–954, 2011.

[76] MARKUTSYA, S. and LAMM, M. H., "A coarse-graining approach for molecular simulation that retains the dynamics of the all-atom reference system by implementing hydrodynamic interactions," *The Journal of Chemical Physics*, vol. 141, no. 17, p. 174107, 2014.

[77] MCWEENY, R., "Some recent advances in density matrix theory," *Rev. Mod. Phys.*, vol. 32, pp. 335–369, Apr 1960.

[78] MELLOR-CRUMMEY, J. and GARVIN, J., "Optimizing sparse matrix-vector product computations using unroll and jam," *Intl J. of High Perf. Comput. Appl.*, vol. 18, pp. 225–236, 2004.

[79] MEREGHETTI, P., GABDOULLINE, R. R., and WADE, R. C., "Brownian dynamics simulation of protein solutions: structural and dynamical properties," *Biophysical Journal*, vol. 99, pp. 3782–3791, 2010.

[80] METROPOLIS, N., ROSENBLUTH, A. W., ROSENBLUTH, M. N., TELLER, A. H., and TELLER, E., "Equation of state calculations by fast computing machines," *The journal of chemical physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[81] MIAO, Y. and MERZ, K. M., "Acceleration of electron repulsion integral evaluation on graphics processing units via use of recurrence relations," *Journal of Chemical Theory and Computation*, vol. 9, no. 2, pp. 965–976, 2013.

[82] MONAKOV, A. and AVETISYAN, A., "Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs," in *Proc. 9th Intl Workshop Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, (Berlin, Heidelberg), pp. 289–297, Springer-Verlag, 2009.

[83] MONAKOV, A., LOKHMOTOV, A., and AVETISYAN, A., "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *Proc. 5th Intl Conf. High Perf. Embedded Architectures and Compilers*, HiPEAC'10, (Berlin, Heidelberg), pp. 111–125, Springer-Verlag, 2010.

[84] NIEPLOCHA, J., HARRISON, R. J., and LITTLEFIELD, R. J., "Global arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[85] NISHTALA, R., VUDUC, R. W., DEMMEL, J. W., and YELICK, K. A., "When cache blocking of sparse matrix vector multiply works and why," *Appl. Algebra Eng., Commun. Comput.*, vol. 18, pp. 297–311, 2007.

[86] O'LEARY, D. P., "The block conjugate gradient algorithm and related methods," *Linear Algebra and Its Applications*, vol. 29, pp. 293–322, 1980.

[87] OLIKER, L., LI, X., HUSBANDS, P., and BISWAS, R., "Effects of ordering strategies and programming paradigms on sparse matrix computations," *SIAM Rev.*, vol. 44, pp. 373–393, 2002.

[88] OLIVIER, S., PORTERFIELD, A., WHEELER, K., and PRINS, J., "Scheduling task parallelism on multi-socket multicore systems," in *Proc. 1st Intl. Workshop Runtime and Operating Systems for Supercomputers*, ROSS '11, pp. 49–56, ACM, 2011.

[89] PALSER, A. H. R. and MANOLOPOULOS, D. E., "Canonical purification of the density matrix in electronic-structure theory," *Phys. Rev. B*, vol. 58, pp. 12704–12711, Nov 1998.

[90] PARKS, M. L., DE STURLER, E., MACKEY, G., JOHNSON, D. D., and MAITI, S., "Recycling Krylov Subspaces for Sequences of Linear Systems," *SIAM J. Sci. Comput.*, vol. 28, pp. 1651–1674, Sept. 2006.

[91] PINAR, A. and AYKANAT, C., "Fast optimal load balancing algorithms for 1D partitioning," *J. Parallel Distrib. Comput.*, vol. 64, pp. 974–996, 2004.

[92] PINAR, A. and HEATH, M. T., "Improving performance of sparse matrix-vector multiplication," in *Proc. ACM/IEEE Conf. Supercomputing*, SC '99, (New York, NY, USA), ACM, 1999.

[93] PLIMPTON, S., "Fast parallel algorithms for short-range molecular-dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, 1995.

[94] POPLE, J. A., GILL, P. M., and JOHNSON, B. G., "Kohn-Sham density-functional theory within a finite basis set," *Chemical Physics Letters*, vol. 199, no. 6, pp. 557–560, 1992.

[95] PUTZ, V. B., DUNKEL, J., and YEOMANS, J. M., "CUDA simulations of active dumbbell suspensions," *Chemical Physics*, vol. 375, pp. 557–567, 2010.

[96] RAHMAN, A., "Correlations in the motion of atoms in liquid argon," *Physical Review*, vol. 136, no. 2A, p. A405, 1964.

[97] RAMDAS, T., EGAN, G. K., ABRAMSON, D., and BALDRIDGE, K. K., "On ERI sorting for SIMD execution of large-scale Hartree-Fock SCF," *Computer Physics Communications*, vol. 178, no. 11, pp. 817–834, 2008.

[98] RAMDAS, T., EGAN, G. K., ABRAMSON, D., and BALDRIDGE, K. K., "ERI sorting for emerging processor architectures," *Computer Physics Communications*, vol. 180, no. 8, pp. 1221–1229, 2009.

[99] ROTNE, J. and PRAGER, S., "Variational treatment of hydrodynamic interaction in polymers," *Journal of Chemical Physics*, vol. 50, pp. 4831–4837, 1969.

[100] RYS, J., DUPUIS, M., and KING, H. F., "Computation of electron repulsion integrals using the Rys quadrature method," *Journal of Computational Chemistry*, vol. 4, no. 2, pp. 154–157, 1983.

[101] SAINTILLAN, D., DARVE, E., and SHAQFEH, E. S., "A smooth particle-mesh Ewald algorithm for Stokes suspension simulations: The sedimentation of fibers," *Physics of Fluids*, vol. 17, pp. 033301–033301, 2005.

[102] SCHMIDT, M. W., BALDRIDGE, K. K., BOATZ, J. A., ELBERT, S. T., GORDON, M. S., JENSEN, J. H., KOSEKI, S., MATSUNAGA, N., NGUYEN, K. A., SU, S., and OTHERS, "General atomic and molecular electronic structure system," *Journal of Computational Chemistry*, vol. 14, no. 11, pp. 1347–1363, 1993.

[103] SCHUBERT, G., HAGER, G., FEHSKE, H., and WELLEIN, G., "Parallel Sparse Matrix-Vector Multiplication as a Test Case for Hybrid MPI+OpenMP Programming," in *IPDPS Workshops*, 2011.

[104] SIEROU, A. and BRADY, J. F., "Accelerated Stokesian dynamics simulations," *Journal of Fluid Mechanics*, vol. 448, pp. 115–146, 2001.

[105] SIEROU, A. and BRADY, J. F., "Accelerated Stokesian Dynamics simulations," *Journal of Fluid Mechanics*, vol. 448, pp. 115–146, 2001.

[106] SOLOMONIK, E., MATTHEWS, D., HAMMOND, J., and DEMMEL, J., "Cyclops Tensor Framework: reducing communication and eliminating load imbalance in massively parallel contractions," in *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[107] STROUT, D. L. and SCUSERIA, G. E., "A quantitative study of the scaling properties of the Hartree-Fock method," *The Journal of Chemical Physics*, vol. 102, p. 8448, 1995.

[108] SU, B. and KEUTZER, K., "clSpMV: A cross-platform OpenCL SpMV framework on GPUs," in *Proc. 26th Intl Conf. Supercomputing*, ICS '12, pp. 353–364, ACM, 2012.

[109] SZABO, A. and OSTLUND, N. S., *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. Dover, 1989.

[110] TORRES, F. E. and GILBERT, J. R., "Large-Scale Stokesian Dynamics Simulations of Non-Brownian Suspensions," tech. rep., Xerox Research Centre of Canada, 1996.

[111] UFIMTSEV, I. S. and MARTINEZ, T. J., "Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation," *Journal of Chemical Theory and Computation*, vol. 4, no. 2, pp. 222–231, 2008.

[112] VALIEV, M., BYLASKA, E. J., GOVIND, N., KOWALSKI, K., STRAATSMA, T. P., VAN DAM, H. J., WANG, D., NIEPLOCHA, J., APRA, E., WINDUS, T. L., and OTHERS, "NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010.

[113] VAN DE GEIJN, R. A. and WATTS, J., "SUMMA: scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[114] VÁZQUEZ, F., FERNÁNDEZ, J., and GARZÓN, E., "A new approach for sparse matrix vector product on NVIDIA GPUs," *Concurr. Comput.: Pract. Exper.*, vol. 23, pp. 815–826, 2011.

[115] VIERA, M. N., *Large scale simulation of Brownian suspensions*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[116] VUDUC, R. W. and MOON, H.-J., "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *Proc. High-Perf. Comput. and Commun. Conf.*, HPCC'05, (Sorrento, Italy), pp. 807–816, 2005.

[117] VUDUC, R. W., *Automatic performance tuning of sparse matrix kernels*. PhD thesis, Univ. of California, Berkeley, 2003.

[118] WILKINSON, K. A., SHERWOOD, P., GUEST, M. F., and NAIDOO, K. J., "Acceleration of the GAMESS-UK electronic structure package on graphical processing units," *Journal of Computational Chemistry*, vol. 32, no. 10, pp. 2313–2318.

[119] WILLCOCK, J. and LUMSDAINE, A., "Accelerating sparse matrix computations via data compression," in *Proc. 20th Intl Conf. Supercomputing*, ICS '06, pp. 307–316, ACM, 2006.

[120] WILLIAMS, S., OLIKER, L., VUDUC, R., SHALF, J., YELICK, K., and DEMMEL, J., "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proc. ACM/IEEE Conf. Supercomputing*, SC '07, (New York, NY, USA), pp. 38:1–38:12, ACM, 2007.

[121] WILLIAMS, S. W., *Auto-tuning performance on multicore computers*. PhD thesis, Univ. of California, Berkeley, 2008.

[122] YAMAKAWA, H., "Transport properties of polymer chains in dilute solution: Hydrodynamic interaction," *Journal of Chemical Physics*, vol. 53, pp. 436–443, 1970.

[123] YASUDA, K., "Two-electron integral evaluation on the graphics processor unit," *Journal of Computational Chemistry*, vol. 29, no. 3, pp. 334–342, 2008.

[124] ZIMMERMAN, S. B. and MINTON, A. P., "Macromolecular crowding - biochemical, biophysical, and physiological consequences," *Annual Review of Biophysics and Biomolecular Structure*, vol. 22, pp. 27–65, 1993.

[125] ZIMMERMAN, S. B. and TRACH, S. O., "Estimation of macromolecule concentrations and excluded volume effects for the cytoplasm of Escherichia-coli," *Journal of Molecular Biology*, vol. 222, no. 3, pp. 599–620, 1991.

VITA

Xing Liu is a PhD student in Computational Science and Engineering at Georgia Institute of Technology, advised by Prof. Edmond Chow. Before Xing started his PhD in August 2009, he was a staff R&D engineer at IBM (Shanghai, China, from 2006 to 2009). Xing obtained the M.Eng. and B.Eng. degrees in Electronic and Information Engineering from Huazhong University of Science and Technology, in 2003 and 2006, respectively.

Xing's research mainly focuses on the invention of novel parallel numerical and discrete algorithms for solving real world scientific and engineering problems on emerging high performance architectures, including high-end multicore processors and accelerators. He is particularly interested in the areas of sparse linear algebra, computational biology, quantum chemistry, and large-scale physics-based simulations.

Xing is a member of ACM, IEEE, and SIAM.